Nutzung des Domänenmodells zur leichteren Spezifizierung von SQL-Anfragen im Finanzinformationssystem eHÜL

Felix Seggebäing

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik, Abteilung III

Vorgelegt von:
Felix Seggebäing
Matrikelnr.: 3281203
E-Mail: `seggebaeing@uni-bonn.de`

Erstgutachter:
Dr. Thomas Bode
Zweitgutachter:
Prof. Dr. Rainer Manthey

Ich möchte meinem Betreuer Dr. Thomas Bode herzlich danken für die Unterstützung und die wertvollen Hinweise bei der Erstellung dieser Arbeit.
Weiter gilt mein tiefer Dank meinen Eltern, die mich in allen Belangen immer bedingungslos unterstützt haben, insbesondere meinem Vater für seine Hilfe beim Korrekturlesen und seine Unterstützung während der Bachelorarbeit.

Im Rahmen dieser Arbeit wird das generische Maskulinum verwendet. Ich weise ausdrücklich darauf hin, dass dort alle Geschlechtsidentitäten einbezogen sind.

Inhaltsverzeichnis

1 Einleitung 1
2 Grundlagen 2
2.1 eHÜL 2
BDOs 2
2.2 QRL 3
2.3 JPA 5
Hibernate 5
Das Metamodell 6
3 Problemstellung und Anforderungen 6
3.1 Aktueller Stand und Probleme 6
Vertikale Partitionierung 7
Namenskonventionen und begriffliche Abweichungen 7
Vererbung und polymorphe Strukturen 7
Many-to-Many-Zuordnungen und Join-Tables 8
Implikationen für die Generierung von SQL-Abfragen 8
3.2 Zielsetzung und Anforderungen 8
Funktionale Anforderungen 9
Nicht-funktionale Anforderungen 9
Gelöste Probleme und Herausforderungen im Fokus 10
Ausblick 10
4 Lösungsansätze und Entwurf 10
4.1 Ansätze zur Umsetzung der Generierung 11
Regelbasierte Generierung 11
Verwendung domänenmodellbasierter Abfragesprachen (JPQL, HQL) 11
LLM-gestützte Generierung 11
4.2 Architektur der Komponente 12
Eingabefeld für den Prompt 12
Eingabefeld für das resultierende SQL-Statement 13
Kontextauswahl (betroffene Domänentypen) 13
Hilfefenster mit Tipps zur Formulierung 13
Knopf zur Fehlerüberprüfung 14
4.3 Kontextgenerierung 14
Domänenmodell 14
Datenbankschema 14
4.4 Fehlerbehandlung 16
5 Ausgewählte Aspekte der Implementierung 17
5.1 Kontextgenerierung 17
Verwaltung der Auswahl 17
Extraktion des JPA-Metamodells 18
Ableitung des Datenbankschemas 18

5.2 Hilfestellungen für den Nutzer 19
6 Evaluation 20
6.1 Einflussfaktoren auf die Generierung 20
Modellwahl 20
Formulierungsarten von Nutzerprompts 21
Komplexität der SQL-Abfragen 21
6.2 Auswertung 22
6.3 Ergebnisse 27
7 Ausblick und Fazit 33
7.1 Ausblick 33
7.2 Fazit 34
A *Auswertungen* aus dem Produktivsystem 38
A.1 Buchungen bestimmter Finanzierungen 38
Formulierungsvarianten des Prompts 38
A.2 Sachbuchungen mit Rechnungs- und PSP-Daten 39
Formulierungsvarianten des Prompts 39
A.3 Finanzierungen mit vollständigen Daten 39
Formulierungsvarianten des Prompts 40
A.4 Kostengruppen mit zugehörigen PSP-Elementen 41
Formulierungsvarianten des Prompts 41
A.5 Personalmittel bestimmter Personen 41
Formulierungsvarianten des Prompts 42
A.6 Personalmittelfinanzierungen 42
Formulierungsvarianten des Prompts 43
A.7 Betragssummen für Sachbuchungen 44
Formulierungsvarianten des Prompts 44
A.8 Constraint: Budget der Finanzierung 44
Formulierungsvarianten des Prompts 44
A.9 Problembuchungen 45
Formulierungsvarianten des Prompts 46
B Verwendete R-Skripte 47
B.1 Boxplot der LLMs 47
B.2 Boxplot der Komplexitäten 47
B.3 Boxplot der Promptformulierungsarten 48
B.4 Heatmap nach LLM und Komplexität 48
B.5 Heatmap nach Komplexität und Promptformulierungsart 49
B.6 Heatmap nach LLM und Promptformulierungsart 50
B.7 Facettiertes Säulendiagramm 50

1 Einleitung

Die effiziente Verwaltung und Auswertung von Daten stellt eine wichtige Aufgabe moderner Informationssysteme dar. Dies gilt auch für das Finanzinformationssystem *elektronische Haushaltsüberwachungsliste* (eHÜL), das am Institut für Informatik der Universität Bonn entwickelt wurde und seit vielen Jahren kontinuierlich erweitert wird. Das System stellt eine flexible und leistungsstarke Plattform zur Übersicht und Überwachung von Finanzdaten zur Verfügung.[1]

Ein wesentliches Problem vieler Datenbankanwendungen besteht im sogenannten objektrelationalen Mismatch, so auch hier: Die strukturellen Unterschiede zwischen dem objektorientierten Domänenmodell der eHÜL und dem relationalen Schema der zugrunde liegenden Datenbank erschweren die Abbildung und Manipulation von Daten. Im Großen löst das Framework *Hibernate* dieses Problem in der eHÜL. Wenn Nutzer nun aber freie Abfragen an die Datenbank stellen wollen, müssen Sie jedoch SQL schreiben. Hierzu ist eine Kenntnis des genauen relationalen Schemas der Datenbank vonnöten, die meisten Nutzer sind jedoch nur mit dem Domänenmodell der eHÜL vertraut.

Das Ziel dieser Arbeit ist die Entwicklung eines Mechanismus, der es Nutzern ermöglicht, SQL-Abfragen direkt über dieses anwendungsseitige Objektmodell zu formulieren. Um dieses Ziel zu erreichen, sind verschiedene Ansätze denkbar. Im Rahmen dieser Arbeit werden verschiedene Ansätze betrachtet, ihre Vor- und Nachteile gegeneinander aufgewogen und schließlich ein LLM-gestützter Ansatz verfolgt.

Bei dem gewählten Ansatz werden die SQL-Abfragen auf Grundlage einer textuellen Beschreibung sowie strukturiertem Kontext zum Datenmodell der Anwendung und der Datenbank automatisch generiert. Da dieser Ansatz mit verschiedenen Parametern einhergeht (Modellwahl, Promptformulierung, Komplexität des geforderten Statements) werden die Möglichkeiten empirisch untersucht und die Ergebnisse präsentiert. Im Weiteren werden Schwierigkeiten und Limitierungen des Ansatzes diskutiert. Final mündet der Mechanismus in einem grafischen Fenster der eHÜL.

Die vorliegende Arbeit beginnt mit einer Darstellung für die Erweiterung relevanter Grundlagen und Anforderungen. Anschließend werden verschiedene Lösungsansätze entwickelt und diskutiert, um eine technisch und konzeptionell sinnvolle Implementierung zu erarbeiten. Zuletzt wird die Implementierung des Entwurfs beschrieben.

Durch die Ausarbeitung eines benutzerfreundlichen Werkzeugs trägt diese Arbeit zur Weiterentwicklung des eHÜL-Systems bei und bietet zugleich Ansätze für die Gestaltung moderner Datenbankanwendungen in Zeiten aufstrebender generativer Sprachmodelle.

2 Grundlagen

Dieses Kapitel stellt die Grundlagen vor, die für das Verständnis der Arbeit notwendig sind. Zunächst wird näher auf das betrachtete Datenbankanwendungsprogramm, die *elektronische Haushaltsüberwachungsliste* (eHÜL) eingegangen. Dort werden auch die bereits konzipierten und implementierten Teile des Programms im genaueren beschrieben, auf denen diese Arbeit aufbaut und welche hier erweitert werden sollen. Folgend wird noch die Bibliothek zur Datenbankkommunikation *Hibernate* erläutert, auf denen die eHÜL basiert.

2.1 eHÜL

Die *elektronische Haushaltsüberwachungliste* (eHÜL) ist ein am Institut für Informatik der Universität Bonn entwickeltes Finanzinformationssystem, das unter der Leitung von Herrn Dr. Thomas Bode steht und unter anderem im Rahmen von Projektgruppen und Abschlussarbeiten erweitert wird. Es arbeitet auf einer *PostgreSQL*-Datenbank.

Da das Programm über die Jahre erheblich an Komplexität und Umfang gewonnen hat, werden im Folgenden nur die für diese Arbeit relevanten Bereiche kurz beschrieben.

BDOs

Die Geschäftsobbjekte (*business domain objects, BDOs*) der eHÜL bilden ein komplexes Geflecht, wie das (vereinfachte) UML-Diagramm in Abb. 1 verdeutlicht. Jede BDO-realisierende Klasse stellt einen eigenen Entitätstypen dar und wird entsprechend von Hibernate auf die Datenbank abgebildet. Im wesentlichen handelt es sich bei den anwendungsseitigen Klassen um Repräsentationen von Konzepten aus der Realität und stammen aus dem Kontext der Finanzverwaltung bzw. -information.

Im Ausschnitt ist folgendes dargestellt: Personen haben Beschäftigungsverhältnisse (kurz BV, Arbeitsverträge) mit dem Institut, die in Abschnitte unterteilt sind. So kann zu verschiedenen Zeiten eine andere Lohnstufe oder eine veränderte Stundenzahl vorliegen. Die genaue Information über die Finanzierung der BVs stammen aus entsprechenden Objekten, die ihre Deckung aus PSP-Elementen (Konten) erfahren, diese wiederum sind in Kostengruppen organisiert. Auch Rechnungen, die mit Kosten einhergehen und zur Begleichung weiterer, externer Buchungen eingesetzt werden, sind so verknüpft. Aufgrund dieser Tatsache erben Rechnungen und Finanzierung beide von der Klasse *Personal-Einname-Rechnung-Objekt*, kurz PEAO. Sie beide verfügen über ein Budget. Budgets wiederum sind in mehrere Teilbudgets unterteilt, die vom Typ Finanz- oder Personalbudget sein können. Eine Form von Budget sind die Qualitätsverbesserungsmittel von Organisationseinheiten innerhalb des Instituts. Innerhalb dieser Organisationseinheiten liegen die Stellen, die schließlich über

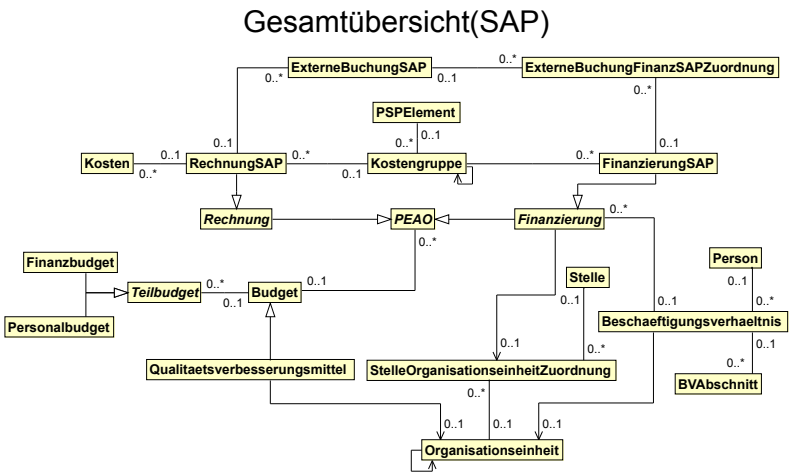

Abb. 1: Ein Ausschnitt des BDM-UML-Diagramm, wie es in dem eHÜL-Wiki zu finden ist.[1]

die bereits beschreibenen BVs besetzt werden und deren Zuordnung wiederum von einer Finanzierung begleitet wird.

Gemäß der objektorientierten Natur der Programmiersprache Java ist auch das BDM objektorientiert formuliert. Klassen beschreiben aus ihnen instanziierte Objekte. In diesen Objekten werden Daten in Attributen gespeichert, Polymorphie und Vererbung werden direkt unterstützt. Um Beziehungen abzubilden, referenzieren Klassen einander direkt. In relationalen Schemata werden Daten hingegen in Tabellen gehalten, Konzepte wie Vererbung müssen über verschiedene Strategien umgesetzt werden und Beziehungen werden über Fremdschlüssel dargestellt.

Auf eine Einführung des genauen relationalen Modells in ähnlicher Ausführlichkeit anhand eines äquivalenten ER-Diagramms wird hier verzichtet, da dies wenig Erkenntnisgewinn bedeuten würde.

2.2 QRL

Die *Query Representation Language* (QRL) bildet eine Repräsentation relationaler Abfragen. Während bekannte *Query Languages* wie SQL die Möglichkeit bieten die Abfragen an sich textuell darzustellen, vermag die QRL Filterbedingungen einer solchen relationalen Abfrage in einer prozedural gut nutzbaren, baumartigen Datenstruktur zu halten. Innerhalb der eHÜL bildet das QRL-System ein dreistufiges System zur Realisierung von Such- und Filtermöglichkeiten über das Datenmodell.

In der ersten und zweiten Stufe kann der Nutzer über spezielle Eingabefelder die Suchergebnisse filtern. Diese sind für die Arbeit jedoch nicht weiter relevant und werden entsprechend auch hier nicht weiter behandelt.

Die dritte Stufe hebt sich von den ersten beiden ab, indem dort eine vollständig freie Eingabe von `SELECT`-Statements in SQL möglich ist. Die Zugehörigkeit zum QRL-System ist somit eine rein formale, die eigentliche Repräsentationssprache ist hier kein Teil der Funktionalität.

Die SQL-Abfragen (im Programm *Auswertung* genannt) repräsentieren eine eigenständiges Domänenobjekt außerhalb des üblichen BDO-Musters. Diese haben keinerlei Beziehungen zu anderen Domänenobjekten aus dem Modell und stellen selber nur eine Kapselung für SQL-Abfragen dar. Zusätzlich zur SQL-Abfrage wird ein Wahrheitswert gespeichert, der angibt, ob die Abfrage in ihrer aktuellen Form ausführbar ist. Dieser Wert wird nach jeder Änderung der Abfrage durch die zugehörige *Service*-Klasse reevaluiert. Techniken um dies durchzuführen, sind bereits implementiert und erprobt.

Darüber hinaus werden sogenannte *ID-Spalten* gespeichert, die vom Nutzer ausgewählt werden und die Primärschlüssel-IDs aus der Datenbank enthalten. Diese ermöglichen es, direkt zu den Detailansichten der mit der Abfrage gefundenen Objekte zu navigieren. Das in der eHÜL bestehende Detailfenster ist in Abb. 2 zu sehen.[1]

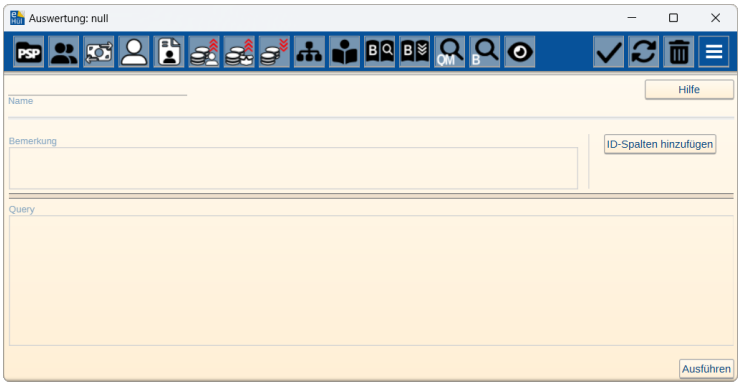

Abb. 2: Detail-Fenster QRL Stufe 3 in der eHÜL, leeres Fenster zwecks Neuanlage.

Die aktive Nutzung der dritten Stufe ist nur Nutzern mit Administratorrechten vorbehalten. Nur diese können Auswertungen speichern und bearbeiten. Somit entfällt ein ausgefeilter Sicherheitsmechanismus, da diese Nutzergruppe ohnehin weitreichende Rechte auf der Datenbank besitzt. Zwecks Validierung bestehen

nur folgende rudimentäre Ansätze:

Nach Prüfung, ob der angemeldete Nutzer Administratorrechte hält, wird zunächst sichergestellt, dass die Abfrage tatsächlich mit einem „`SELECT` “ beginnt und kein Semikolon enthält. Folgend wird nur noch auf Vorhandensein bestimmter, als kritisch eingestufter Schlüsselwörter (`INSERT`, `DELETE`, `CALL`, `DROP`, `SELECT INTO`) geprüft.

In der aktuellen Implementierung sind in dem Detailfenster somit nur `SELECT`-Statements zulässig. Diese können auch parametrisiert werden. Wird die Abfrage vom Service als ausführbar evaluiert, kann ein Ergebnisfenster geöffnet werden. In diesem erscheint eine Tabellenansicht, die alle spezifizierten Spalten enthält, sowie zu den Parametern passende Eingabefelder. Nach Setzen der Werte kann die Auswertung ausgeführt werden. In der Tabellenansicht erscheint in der Folge das Ergebnis.

2.3 JPA

Die *Jakarta Persistence API* (früher *Java Persistence API*), kurz JPA, ist eine Java-Bibliothek die eine Schnittstelle zur objektrelationalen Abbildung (engl. object-relational mapping, ORM) von objektorientierten Daten auf eine Datenbank bereitstellt. Neben der Verwaltung der Datenbankverbindung löst sie noch den sog. Impedance Mismatch auf (dt. etwa *objektrelationale Unverträglichkeit*). Dies beschreibt die konzeptionellen Unterschiede zwischen dem objektorientierten Paradigma, das in Programmiersprachen wie Java verwendet wird, und dem relationalen Modell, auf dem Datenbanken basieren. Hierbei geht es darum, Daten aus der objektorientierten Domäne bijektiv in die relationale Datenbank abzubilden und dabei die konzeptuellen Unverträglichkeiten der beiden Modelle zu umgehen oder aufzulösen. Sie dient somit der Vereinfachung der Übertragung und Zurechnung von Java-Objekten in eine Datenbank, sowie dem Auslesen derselben in die andere Richtung.[2]

In dem Paket gibt es eine Reihe von Java-Annotationen, die genutzt werden, um die einzelnen objektorientierten Komponenten den entsprechenden relationalen Gegenstücken zuzuordnen (Klassen zu Tabellen, Attribute zu Spalten, Objektreferenzen zu Fremdschlüsseln, ...). Auch die verschiedenen Arten der objektorientierten Vererbung lassen sich hier für eine in das relationale Schema übertragene Hierarchie setzen und nutzen.[2]

Hibernate

Hibernate ist ein Framework, das die JPA implementiert. Dabei nutzt es einen *Java-Database-Connector-Treiber* (JDBC-Treiber). Wenn man ein Java-Objekt neu erstellt oder ein bereits vorhandenes aus der Datenbank lädt und dieses aktualisiert oder löscht, übernimmt Hibernate die Persistierung. Insbesondere werden auch Beziehungen zwischen diesen Objekten auf der Anwendungsdomäne

in der Datenbank nach relationalem Schema gespeichert.[3]

Die eHÜL nutzt Hibernate für die Persistierung ihrer Daten in der Datenbank.

Das Metamodell

Bei Erstellung einer (beispielsweise Hibernate-) Instanz (genauer: Bei der Initialisierung der *SessionFactory*) legt sich diese im Hintergrund eine Datenstruktur an, welche das BDM in Form eines Beziehungsgraphen modelliert. Hierbei durchsucht Hibernate die in der vom Entwickler angelegten Konfigurationsdatei vermerkten Klassen nach den entsprechenden JPA-Annotationen. In dem Modell werden dann Informationen wie konkrete Abbildungen oder Beziehungskardinalitäten gehalten. Dies geschieht nach dem *„eager-loading“*-Prinzip, bei dem alle benötigten Informationen bereits bei der Initialisierung vollständig geladen werden, um späteren Zugriff zu beschleunigen. Das resultierende Modell heißt *Metamodell*. Es lässt sich durch den Entwickler aus der *SessionFactory* der Hibernate-Instanz laden, falls an anderer Stelle außerhalb der direkten Hibernate-Umgebung ebenfalls Bedarf an diesen Informationen besteht. Diese Funktionalitäten sind Teil der JPA und somit auch in anderen JPA-Implementierungen als Hibernate vorhanden.[3]

3 Problemstellung und Anforderungen

3.1 Aktueller Stand und Probleme

Aktuell erfolgt der Großteil der Datenbankabfragen in der eHÜL über vorgefertigte Formulare und Übersichten. Mithilfe der ersten und zweiten Stufe der QRL können Nutzer ohne Erfahrung mit dem Datenbank-Backend bereits viele Filtermöglichkeiten in den Übersichtsfenstern nutzen. Jedoch gibt es immer wieder Situationen, in denen Benutzer von der Standardfunktionalität abweichen und spezifische Abfragen stellen möchten. Um solche Abfragen zu realisieren, ist es bisher notwendig, direkt SQL zu schreiben oder von extern zu importieren. Dies bringt einige Probleme mit sich, die im Folgenden diskutiert werden.

Die Diskrepanz zwischen objektorientiertem BDM der eHÜL und relationaler Datenbank führt zu einem grundlegenden Problem: dem objektrelationalen Mismatch. Dieser beschreibt die konzeptionellen Unterschiede zwischen der objektorientierten Welt der Anwendungslogik und dem relationalen Modell der Datenbank.

Der Kern des objektrelationalen Mismatch liegt in der unterschiedlichen Strukturierung von Daten in den beiden Welten. Während das Domänenmodell der eHÜL Klassen wie *Personen*, *Finanzierungen* oder *Beschäftigungsverhältnisse* umfasst, werden diese Entitätstypen in der Datenbank als Tabellen mit Primärschlüsseln, Fremdschlüsseln und normalisierten Beziehungen dargestellt.[4]

Hibernate dient hierbei als Brücke, indem es eine objektrelationale Abbildung bereitstellt. Jedoch bildet dies nur die grundlegende Schnittstelle der beiden Modelle. Nutzer, die individuelle und komplexe Abfragen erstellen möchten, müssen direkt SQL schreiben. Hierzu ist jedoch ein detailliertes Wissen über die Struktur der relationalen Datenbank erforderlich, das viele Nutzer, die nur mit dem Domänenmodell vertraut sind, nicht besitzen. Dies erschwert die Erstellung von Abfragen, insbesondere wenn mit komplizierten Beziehungen über mehrere Tabellen gejoint werden muss.

Im Folgenden werden die wesentlichen Unterschiede in der eHÜL konkret betrachtet.

Vertikale Partitionierung

Ein wesentliches Problem ergibt sich aus der fehlenden direkten 1:1-Zuordnung zwischen den Klassen des Domänenmodells und den Tabellen der relationalen Datenbank. Während im Domänenmodell beispielsweise Beschäftigungsverhältnisse direkt als einfache Klasse realisiert sind, wird diese in der Datenbank vertikal partitioniert. So existieren die beiden Tabellen *beschaeftigungsverhaeltniscore* und *beschaeftigungsverhaeltnisdetails*. Um eine vollständige Abfrage zu einem Beschäftigungsverhältnis zu formulieren, muss ein Nutzer also wissen, dass er diese beiden Tabellen verknüpfen muss. Ein analoges Beispiel stellen die PSP-Elemente dar, die sich in der Datenbank auf die Tabellen *pspelementcore* und *pspelementdetails* aufteilen.

Namenskonventionen und begriffliche Abweichungen

Zusätzlich erschwert wird die manuelle SQL-Formulierung durch inkonsistente Bezeichner. Beispielsweise enthält das Domänenmodell den Entitätstypen *Einnahme*, die Datenbank hingegen die zugeordnete Tabelle *einnahmen*. Auch gibt es einige historischen Überbleibsel, welche potenzielle Verwirrungsquellen darstellen. So bildet Hibernate das BDO *Finanzierung* auf die Tabelle *finanzierungsap* ab, daneben gibt es allerdings auch noch die Tabelle *finanzierunghis*, welche in der eHÜL selber gar nicht mehr genutzt wird. Zusätzlich zur abweichenden Benennung der Tabelle gibt es also noch einen weiteren Kandidaten, der rein vom Namen her den Entitätstypen in der Datenbank darstellen könnte, in der die Finanzierungsobjekte persistiert werden.

Vererbung und polymorphe Strukturen

Im Domänenmodell gibt es zahlreiche Fälle von Vererbung, beispielsweise *Personalbudget*, das von *Teilbudget* erbt, oder *Qualitätsverbesserungsmittel*, das eine Unterklasse von *Budget* darstellt. Die relationale Datenbank setzt solche Vererbungsstrukturen durch verschiedene Strategien um. Teilweise wird das *Single-Table-Inheritance*-Pattern genutzt, in anderen Fällen jedoch der *Table-per-Class*-Ansatz gewählt. Nutzer, die auf die Datenbank zugreifen, müssen folglich

nicht nur wissen, welche Tabelle sie anfragen müssen, sondern auch, wie die Vererbung implementiert ist, um die relevanten Daten korrekt zu extrahieren.

Many-to-Many-Zuordnungen und Join-Tables

Ein weiteres Hindernis stellen die Beziehungen innerhalb der Datenmodelle dar. Während im Domänenmodell eine Beziehung einfach durch eine Referenz in Form eines Attributs dargestellt wird, erfordert die relationale Umsetzung oft Zwischentabellen. So hat beispielsweise das BDO *Nutzer* im Domänenmodell eine Beziehung zu *Nutzergruppe*, die in der Datenbank durch die Tabelle *nutzergruppenutzer* realisiert wird. So müssen in SQL also nicht nur die eigentlichen Entitätstypen abgefragt werden, sondern es müssen auch die entsprechende Zwischentabellen gejoint werden, um eine vollständige Abfrage zu erhalten.

Implikationen für die Generierung von SQL-Abfragen

Diese Diskrepanzen erschweren die Generierung von SQL-Abfragen aus dem Domänenmodell heraus. Eine einfache Übersetzung von Klassen und Attributen in SQL reicht offensichtlich nicht aus, da Tabellennamen abweichen, Daten auf verschiedene Tabellen verteilt sind, Relationen über Fremdschlüssel oder Zwischentabellen hergestellt werden müssen und Typkonvertierungen erforderlich sind. Somit benötigen ausschließlich mit dem Domänenmodell vertraute Nutzer ein Werkzeug, das ihnen diese Übersetzung vom Domänenmodell in das relationale Modell abnimmt. Ohne ein erweitertes Verständnis der relationalen Struktur lassen sich keine korrekten und effizienten SQL-Abfragen formulieren.

3.2 Zielsetzung und Anforderungen

Um diese Probleme zu lösen, wird ein Mechanismus benötigt, der es erlaubt, SQL-Abfragen über das Domänenmodell intuitiv zu formulieren. Der Mechanismus muss die Beziehungen und Konzepte des Domänenmodells nutzen, um die Abbildung zum relationalen Schema zu abstrahieren. Der Fokus liegt hierbei auf der Entwicklung der zugrunde liegenden Logik, die das Domänenmodell als Ausgangspunkt für Abfragen nutzt. Dies soll eine intuitive Benutzeroberfläche leisten, die der Visualisierung und Ergänzung dient.

Mit diesem Ansatz sollen die soeben diskutierten Probleme überwunden und die Benutzerfreundlichkeit der eHÜL für die betroffenen Nutzergruppen nachhaltig verbessert werden. Die Effizienz wird gesteigert und Barrieren werden abgebaut.

Die Entwicklung eines Mechanismus zur vereinfachten Erstellung von SQL-Abfragen in der eHÜL erfordert eine Definition der funktionalen und nichtfunktionalen Anforderungen. Diese Anforderungen basieren auf den in der Problemstellung identifizierten Herausforderungen.

Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die spezifischen Fähigkeiten, die der Mechanismus erfüllen muss.

- **Unterstützung des Domänenmodells:** Der Mechanismus muss die Erstellung von Abfragen auf Basis des objektorientierten Domänenmodells ermöglichen.
- **Automatische Abbildung auf das relationale Schema:** Es soll eine automatische Übersetzung von Abfragen im Domänenmodell in SQL-Befehle erfolgen.
- **Unterstützung komplexer SQL-Konstrukte:** Der Mechanismus soll Joins über mehrere Tabellen, verschachtelte Abfragen sowie komplexe Bedingungen unterstützen.
- **Grafische Benutzeroberfläche:** Der Query-Builder soll eine intuitive grafische Benutzeroberfläche bereitstellen.
- **Validierung von Abfragen:** Es muss eine automatische Validierung der erstellten Abfragen erfolgen, um sicherzustellen, dass die Syntax korrekt ist und die Abfragen auf der Datenbank ausgeführt werden können.
- **Speicherung und Wiederverwendung von Abfragen:** Abfragen, die von Nutzern erstellt wurden, sollen gespeichert und später wiederverwendet werden können.
- **Markierung von ID-Spalten:** Der Mechanismus soll ermöglichen, bestimmte Spalten in der Abfrage als ID-Spalten zu markieren. Dadurch können die haltenden Objekte direkt aus der Ergebnismenge heraus in einem zugehörigen Detailfenster geöffnet werden.

Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen legen fest, welche Qualitätskriterien der Mechanismus und der Query-Builder erfüllen müssen:

- **Benutzerfreundlichkeit:** Die Benutzeroberfläche muss klar strukturiert und intuitiv bedienbar sein.
- **Performanz:** Die automatische Übersetzung von Domänenabfragen in SQL und die Ausführung der Abfragen müssen effizient erfolgen.
- **Erweiterbarkeit:** Das System soll modular aufgebaut sein, um zukünftige Anpassungen und Erweiterungen zu erleichtern.
- **Fehlertoleranz:** Der Mechanismus soll robuste Validierungen enthalten, um fehlerhafte Eingaben zu erkennen und Rückmeldungen an den Nutzer zu geben.
- **Kompatibilität:** Der Mechanismus muss nahtlos in die bestehende eHÜL-Infrastruktur integriert werden können.

Gelöste Probleme und Herausforderungen im Fokus

Einige Anforderungen sind durch die bestehende dritte Stufe des QRL-Systems bereits implementiert.

Bei den funktionalen Anforderungen sind dies die Speicherung bzw. Wiederverwendbarkeit der Abfragen, die Markierung der ID-Spalten sowie die Validierung. So gibt es eine rudimentäre Prozedur zur Validierung von Abfragen, wie bereits in den Grundlagen zur dritten Stufe des QRL-Systems erläutert wurde.

Bei den nicht-funktionalen Anforderungen ist es im Wesentlichen die Performanz die bereits gelöst wurde. Da in dieser Arbeit primär die Logik und das Frontend betrachtet werden und Performanzprobleme in Bezug auf die Datenbank somit keine Rolle spielen, reduziert sich dies auf deren Effizienz.

Somit bleiben die zentralen Anforderungen, die die wesentlichen Herausforderungen mit sich bringen, folgende:

- **Abstraktion des relationalen Schemas:** Die automatische Abbildung des Domänenmodells auf das relationale Schema muss flexibel genug sein, um unterschiedliche Anwendungsfälle abzudecken.
- **Optimierung komplexer Abfragen:** Der Mechanismus muss in der Lage sein, Abfragen mit mehreren Joins und mit verschachtelter Struktur effizient zu generieren und auszuführen.
- **Visualisierung:** Die grafische Benutzeroberfläche muss eine klare Darstellung des Mechanismus ermöglichen, um die intuitive Bedienung zu ermöglichen.

Ausblick

Die Arbeit konzentriert sich auf die Entwicklung der Übersetzungslogik und deren Integration in die grafische Benutzeroberfläche. Die Anforderungsdefinitionen zielen darauf ab, dass der entwickelte Mechanismus den identifizierten Problemen gerecht wird und dieser eine spürbare Verbesserung der Abfragestellung im eHÜL-System erreicht. Im Folgenden werden auf Basis der soeben spezifizierten Anforderungen Lösungsansätze diskutiert und ein konkreter Entwurf ausgearbeitet.

4 Lösungsansätze und Entwurf

In diesem Kapitel werden zunächst Lösungen der identifizierten Probleme vorgestellt. Folgend wird auf Basis dieser Ansätze diskutiert, welche sich konkret für die zu entwerfende Systemkomponente anbieten. Final wird ein Entwurf erarbeitet und vorgestellt, der im Folgenden Kapitel in die Implementierung mündet.

4.1 Ansätze zur Umsetzung der Generierung

Ziel ist, aus einer abstrakten Beschreibung einer Abfrage auf Basis des Domänenmodells einen funktionsfähigen SQL-Ausdruck zu generieren, der natürlich gemäß seiner Natur auf einem relationalen Datenmodell arbeitet. Somit muss zunächst geklärt werden, welcher Ausgangspunkt gewählt wird, also wie diese abstrakte Beschreibung auf dem Domänenmodell aussehen könne und wie eine Übersetzung aus dieser in SQL aussähe.

Regelbasierte Generierung

Eine erste mögliche Lösung besteht in einem regelbasierten Parser, der basierend auf einer grafischen Nutzereingabe (Query-Builder) syntaktisch korrekte SQL-Ausdrücke generiert. Es müssten algorithmisch Regeln festgehalten werden, welche Eingaben vom Nutzer erwartet und wie diese in SQL umgelegt werden. So könnte der Nutzer grafisch einen Graphen aufbauen, der die genaue Struktur der Abfrage wiedergibt. Dieser Graph würde auf dem Domänenmodell arbeiten. Knoten stellen BDOs dar, Kanten Joins, Bedingungen würden sich über Attribute formulieren lassen, etc. Diese Methode ist kontrolliert und deterministisch, dem steht jedoch ein sehr hoher Implementierungsaufwand gegenüber, einerseits zur Auflösung der Unterschiede zwischen den beiden Datenmodellen und weiter im Hinblick auf komplexe SQL-Features wie verschachtelte Abfragen oder Mengenoperationen.

Verwendung domänenmodellbasierter Abfragesprachen (JPQL, HQL)

Alternativ ließe sich eine Abfragesprache wie JPQL oder HQL als Zwischenstufe nutzen; diese Sprachen arbeiten bereits auf dem Domänenmodell. Die Nutzerformulierung der domänenmodellbasierten Abfrage wäre dann über einen grafischen Builder denkbar; die daraus resultierenden Ausdrücke würden dann vom jeweiligen Framework in SQL übersetzt. Die Problematik dieses Ansatzes liegt darin, dass in HQL und JPQL keine reine Übersetzung ohne Ausführung vorgesehen ist. Bei der Nutzung von HQL ließe sich der übersetzte SQL-Ausdruck allerdings abfangen, dargestellt in Abb. 3.[2][3]

Der so resultierende Ausdruck ist allerdings nicht für menschliche Lesbarkeit entworfen, dies spiegelt sich in der von Hibernate verwendeten Benennung von Tabellen und Attributen wieder, als Hilfestellung für den Nutzer ist dieses Verhalten entsprechend nicht wünschenswert. Zudem fehlt eine native grafische Komponente in Hibernate/JPA. Eine eigenständigen Implementierung einer solchen würde wieder das Problem des unverhältnismäßig hohen Aufwandes mit sich bringen.

LLM-gestützte Generierung

Ebenfalls ließen sich LLMs zur direkten Generierung eines SQL-Ausdrucks nutzen. Der Nutzer würde dabei einen Prompt nutzen, welcher folgende Bestandteile erfordert:

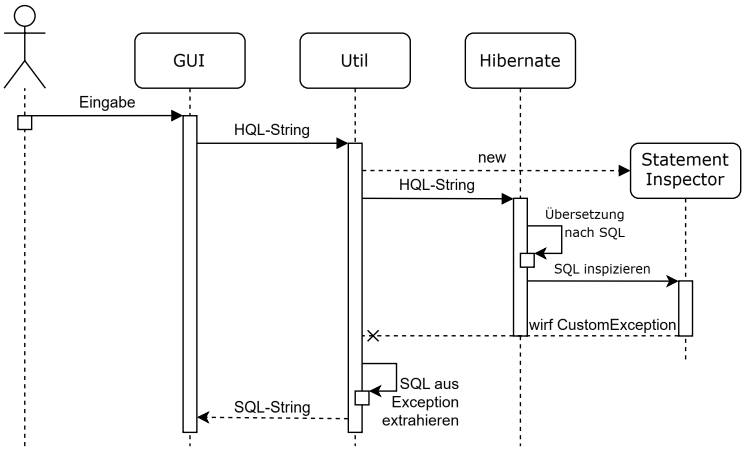

Abb. 3: Sequenzdiagramm: Ein Mechanismus, um in der Ausführung einer HQL-Abfrage das resultierende SQL-Statement abzufangen. Im *StatementInspector* wird eine eigens hierfür angelegte Ausnahme geworfen, in der sich der abrufbare SQL-String befindet. Das Werfen der Exception unterbricht den Programmfluss im Thread und verhindert somit die tatsächliche Ausführung auf der Datenbank, die für gewöhnlich unmittelbar auf die Inspektion folgt.

- eine Anweisung für das Modell,
- eine vom Nutzer formulierten Abfrage in Textform,
- Kontextinformationen: Domänenmodell und Datenbankschema.

Die Vorteile dieses Ansatzes sind in erster Linie der niedrige Implementierungs- und Wartungsaufwand, sowie die hohe Flexibilität, da auch komplexe Konstrukte möglich sind. Ferner sind außerdem die Lerneffekte des Modells nutzbar. Zusätzlich sind stetige Verbesserungen der Modelle zu erwarten, wodurch sich die Funktion ohne Zutun der eHÜL-Entwickler weiter verbessert. Die Nachteile liegen in dem Nichtdeterminismus der Ausgaben, der potenziellen Fehleranfälligkeit und der sich schwer gestaltenden Evaluierung zur Laufzeit.

Nach Abwägung der positiven und negativen Aspekte der einzelnen Ansätze, fällt die Wahl hier auf den LLM-Ansatz. Die Vorteile überwiegen unter Berücksichtigung des Aufwandes klar die Nachteile.

4.2 Architektur der Komponente

Die grafische Komponente ist im Mockup in Abb. 4 dargestellt und besteht aus den folgend beschriebenen Bestandteilen.

Eingabefeld für den Prompt

Hier gibt der Nutzer einen Prompt ein. Dieser Prompt beschreibt zunächst ganz sachlich informell den SQL-Ausdruck, der am Ende resultieren soll. Die verwen-

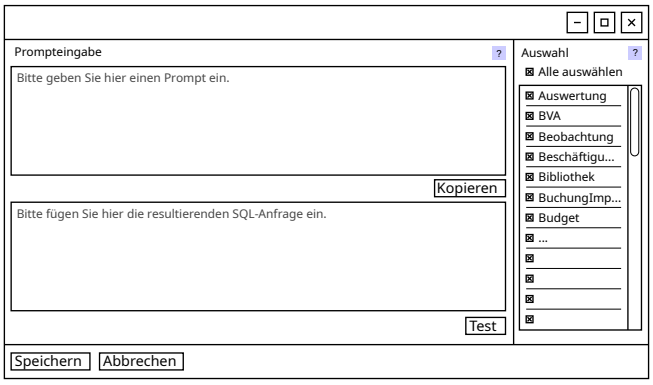

Abb. 4: Mockup: Das Hauptfenster.

deten Begriffe sollen dem Domänenmodell entspringen. Die genaue Formulierung dieses Prompts wird weiter unten noch ausführlicher diskutiert. Weiterer Kontext muss hier durch den Nutzer nicht eingegeben werden, das heißt keine weiteren Anweisungen, was genau vom Modell erwartet wird oder wie das Domänen- oder Datenbankmodell genau aussieht.

Eingabefeld für das resultierende SQL-Statement

Hier fügt der Nutzer nach Generierung durch das LLM den resultierenden Ausdruck ein. Aus diesem Feld heraus ist dann auch ein Syntaxcheck möglich.

Kontextauswahl (betroffene Domänentypen)

Hier kann der Nutzer auswählen, welche Typen aus dem Domänenmodell und somit schließlich auch Entitätstypen aus dem relationalen Schema im Promptkontext dem LLM mit übergeben werden sollen. Es ließe sich zwar auch immer das gesamte Schema übergeben, aber eine Einschränkung auf das Nötige reduziert den Umfang der Nachricht an das Sprachmodell enorm und verbessert somit laut Untersuchungen seine Performanz und Präzision.[5]

Hilfefenster mit Tipps zur Formulierung

Dem Nutzer sollen Fenster zur erleichterten Bedienung zur Verfügung gestellt werden. Einmal soll die Oberfläche an sich erläutert werden, in etwa was die Auswahl betroffener Domänentypen für Konsequenzen hat. Auf der anderen Seite soll Hilfestellung für die Formulierung des Prompts und Auswahl des Sprachmodells geleistet werden.

Knopf zur Fehlerüberprüfung

Nach Einfügen des generierten SQL-Ausdrucks soll dem Nutzer ermöglicht werden, diesen schnell und einfach zu evaluieren. Dies geschieht über einen Knopf, über den der programmatische Evaluierungsvorgang in Bewegung gesetzt wird.

Nach Ausfüllen des Prompts wird der Kontext dynamisch generiert. Zum Kontext gehören jeweils einmal eine detaillierte Beschreibung des Domänenmodells und des Datenbankschemas, bzw. den vom Nutzer als relevant markierten Teilen von diesen. Weiter enthält der Kontext Anweisungen an das LLM, was genau es in der Eingabe zu erwarten hat (ein benutzerdefinierter Prompt aus Domänenmodellbasis, Beschreibungen von Domänen- und Datenbankmodell) und was von ihm erwartet wird (Ausgabe eines PostgreSQL-Ausdrucks und nur dieses Ausdrucks auf Basis der mitgelieferten Beschreibung des Datenbankschemas).

Der Nutzer kann dem kontextualisierten Prompt folgend an das Modell der Wahl übergeben und das Resultat (SQL) in das Programm einfügen. Dieser Zwischenschritt könnte durch eine simple zukünftige Erweiterung, die direkt eine API nutzt, entfallen. Diese ist hier allerdings noch nicht vorgesehen, da die Nutzung der APIs im Regelfall kostenpflichtig ist oder ein Sprachmodell lokal ausgeführt werden muss. Nach optionaler Überprüfung des Ausdrucks kann der Nutzer aus dem Fenster zurückkehren und wie gewohnt mit dem Ausdruck in Form einer eHÜL-Auswertung weiterarbeiten.

4.3 Kontextgenerierung

Die Informationen über die Datenmodelle sollen dynamisch generiert werden. Auf der einen Seite stellt dies die stete Aktualität des Kontextes sicher, auf der anderen Seite lässt sich so durch den Nutzer einschränken, welche Informationen konkret an das Modell übergeben werden.

Domänenmodell

Aus dem Hibernate-Metamodell werden alle Entitätstypen extrahiert. Zu jedem werden Name, Attribute, Beziehungen und ggf. Vererbungen gesammelt. Die Nutzerwahl grenzt diese Daten ein. Die Ausgabe erfolgt in strukturierter JSON-Form zur besseren Verwertbarkeit durch das Sprachmodell.

Datenbankschema

Grundsätzlich bestehen zwei Möglichkeiten, um Informationen über das zugrunde liegende Datenbankschema zu extrahieren:

Eine Option ist die direkte Abfrage der `information_schema`-Tabelle. Diese Variante erlaubt eine präzise Erfassung aller Tabellen, Spalten, Datentypen

und Constraints aus der tatsächlich bestehenden Datenbankinstanz. Sie ist insbesondere dann nützlich, wenn das Schema dynamisch generiert oder manuell verändert wurde. Allerdings ist dieser Ansatz an die Spezifik des jeweiligen SQL-Dialekts gebunden. Die Tabelle gehört zwar zum SQL-Standard[6], jedoch erfüllen einige gängige Implementierungen wie Oracle diese Schnittstelle nicht.[7] Zum aktuellen Zeitpunkt ist keine Migration der eHÜL auf eine andere Datenbankarchitektur geplant, dennoch könnte dies zukünftig zu Kompatibilitätseinschränkungen führen.

Eine andere Möglichkeit stellt die Ableitung aus den Annotationen der JPA-Entitätsklassen dar. Hierbei wird das relationale Schema aus den vorhandenen Annotationen in den Java-Klassen rekonstruiert. Informationen über Tabellen- und Spaltennamen, Datentypen, Schlüssel, Constraints und Beziehungen lassen sich direkt aus Annotationen der JPA extrahieren.[2]

Für diese Arbeit wurde die zweite Variante gewählt. Der Zugriff auf die Annotationen erfolgt im Rahmen derselben Analyse, mit der auch das Domänenmodell zur Kontextgenerierung extrahiert wird. Dadurch entsteht kein großer zusätzlicher Aufwand. Zudem erhöht dieser Weg die Portabilität des Systems, da keine Abhängigkeit vom konkreten SQL-Dialekt besteht. Die gewonnenen Informationen werden analog zum Domänenmodell in strukturierter JSON-Form bereitgestellt. Für Finanzierungen aus dem Domänenmodell resultiert beispielsweise der folgende String:


```
    "entityTypeName": "Finanzierung",
    "superclassEntity": "PEAO",
    "inheritanceType": "TABLE_PER_CLASS",
    "simpleAttributes": [
      {
        "attrName": "abteilungpmb",
        "attrCategory": "singular",
        "attrType": "String"
      },
...
      {
        "attrName": "uks04",
        "attrCategory": "singular",
        "attrType": "String"
      }
    ],
    "relationshipAttributes": [
      {
        "attrName": "bv",
        "attrCategory": "singular",
        "attrType": "Beschaeftigungsverhaeltnis",
        "isMappingSideInDB": true
```

```
      },
 ...
      {
        "attrName": "kostengruppe",
        "attrCategory": "singular",
        "attrType": "Kostengruppe",
        "isMappingSideInDB": true
      }
    ]
```

Das Analogon aus dem relationalen Datenbankmodell wie folgt:

```
"finanzierungsap": {
 "tableName": "finanzierungsap",
        "columns": [
          {
            "name": "anteilvz",
            "type": "DECIMAL",
            "nullable": true,
            "unique": false,
            "length": 255,
            "primaryKey": false
          },
          ...
          {
            "name": "uks04",
            "type": "VARCHAR",
            "nullable": true,
            "unique": false,
            "length": 255,
            "primaryKey": false
          }
        ]
      }
```

Um die Auswahl auf die Nutzerauswahl zu reduzieren, wird der Funktion, die die JSON-Strings generiert, schlicht eben jene Untermenge der Entitätstypen aus dem Metamodel übergeben, die der Nutzer in der GUI zuvor selektiert hat.

4.4 Fehlerbehandlung

Bisher wird lediglich geprüft, ob die gegebene Abfrage tatsächlich mit einem „`SELECT` “ beginnt, kein als problematisch identifiziertes Schlüsselwort, sowie kein Semikolon enthält und tatsächlich ausführbar ist. Sollte eines dieser Kriterien der Ausführung, beziehungsweise der Speicherung der Auswertung im Wege stehen, wird der Nutzer entsprechend benachrichtigt. Nun gesellen sich

zu diesen noch syntaktische und semantische Fehler, die aus dem Sprachmodell selber hervorgehen.[8]

Syntaktisch fehlerhafte SQL-Ausdrücke, die durch das Sprachmodell generiert wurden, werden spätestens beim Speichern oder Testausführen erkannt. Die Validierung greift hier auf den bereits vorhandenen Mechanismus zurück: Die Abfrage wird testweise gegen die Datenbank ausgeführt, ein etwaiger Fehler wird abgefangen. Die von der Datenbank zurückgegebene Fehlermeldung kann extrahiert und – falls gewünscht – dem LLM erneut mitgegeben werden, um eine korrigierte Version zu erzeugen.

Zur Behandlung dieser erkennbaren Fehler wird die testweise Ausführung der generierten SQL-Abfrage im System selbst genutzt. Sollte eine Fehlermeldung resultieren, kann der Nutzer den Fehlertext kopieren und diesen zusammen mit dem ursprünglichen Prompt erneut an das Modell übergeben beziehungsweise die noch vorhandene Konversation nutzen. So entsteht eine einfache, interaktive Fehlerbehebungsschleife, die ohne tiefergreifende technische Eingriffe auskommt.

Auf der anderen Seite sind semantische Fehler nicht ohne weiteres automatisch erkennbar. Sie umfassen beispielsweise eine inkorrekte Join-Logik, zu breite Ergebnismengen oder inkonsistente Bedingungen. Diese Fehler lassen sich nicht durch die Datenbank erkennen, da die Abfragen formal korrekt sind. Entsprechend muss der Nutzer manuell gegenlesen, um das Resultat auf semantische Korrektheit zu prüfen.

5 Ausgewählte Aspekte der Implementierung

5.1 Kontextgenerierung

Die Kontextgenerierung bildet die Grundlage für die automatische Generierung von SQL-Abfragen mithilfe von LLMs. Sie besteht aus zwei Teilen: der Extraktion des JPA-Metamodells und der Ableitung des Datenbankschemas. Dies geschieht in der neu angelegten Hilfsklasse `AuswertungUtil`.

Verwaltung der Auswahl

Um die benutzerdefinierte Auswahl der relevanten Entitätstypen aus dem Business-Domain- und Datenbank-Modell zu verwalten, wurde die innere Klasse `ContextSelectionObject` in der `AuswertungUtil` angelegt. Diese verwaltet eine dynamisch aktualisierte und an `Property`-Objekte aus der GUI gebundene Auswahl von Entitätstypen aus dem Domänenmodell und ist nach außen sichtbar. Weiter greift sie auf alle weiteren privaten Hilfsmethoden der äußeren Klasse zu, somit sind diese sinnvoll gekapselt.

Extraktion des JPA-Metamodells

Zur Extraktion des JPA-Metamodells wird das Metamodell genutzt, um sämtliche verfügbaren Klassen aus dem Domänenmodell zu beziehen. Die erhaltenen BDOs werden im Anschluss iterativ analysiert.

Für jedes BDO werden folgende Informationen extrahiert:

- Name des BDOs
- Attribute mit Name, Datentyp und Kategorie (singulär oder plural)
- Beziehungen zu anderen BDOs einschließlich Kardinalität und Zuordnung
- Vererbungshierarchien und verwendete Inheritancestrategie (Single Table, Joined, Table per Class)

Die Attribute werden dabei nach simplen und relationalen Attributen getrennt behandelt. Alle extrahierten Daten werden in einer strukturierten JSON-Repräsentation gespeichert. Diese JSON-Daten werden anschließend Teil des LLM-Prompts.

Ableitung des Datenbankschemas

Zur Ableitung des Datenbankschemas erfolgt eine separate Analyse der JPA-Annotationen der Entitätsklassen. Diese Analyse nutzt die Java-Reflection-API.

Folgende Informationen werden für jeden Entitätstypen gesammelt:

- Primärtabellenname (aus `@Table`)
- Sekundärtabellen (aus `@SecondaryTable`, respektive `@SecondaryTables`)
- Tabellenspalten (aus `@Column`), inklusive Datentyp, Name, Constraints
- Primärschlüsselspalten (aus `@Id`)
- Join-Tabellen (aus `@JoinTable`) mit ihren teilnehmenden Tabellen, sowie Join-Spalten

Join-Tabellen werden aus der `@JoinTable`-Annotation ausgelesen. Diese hat in der JPA die Eigenschaft, dass sie maximal zwei teilnehmende Entitätstypen-Tabellen hält, im Falle eines Selbstjoins sogar lediglich eine. Beziehungen, die über JoinTables mit mehr als zwei Teilnehmern dargestellt werden, werden selber bereits wieder als Entitätstyp behandelt.[2] So reicht es aus zu prüfen, ob beide teilnehmenden Tabellen (oder im Falle eine Selbstjoins nur die eine) des betrachteten JoinTables in der aktuell relevanten Auswahl liegen. Nur in diesem Fall werden sie in die Kontextinformationen aufgenommen.

Die Ergebnisse der Analyse werden ebenfalls in einer strukturierten JSON-Repräsentation gespeichert und bilden den zweiten Teil des Kontexts, der dem LLM übergeben wird.

Beide generierten JSON-Strukturen – Metamodell und Datenbankschema – werden schließlich zusammengeführt und zusammen mit dem vom Nutzer eingegebenen Prompt an das verwendete LLM weitergereicht, um eine SQL-Abfrage zu generieren.

5.2 Hilfestellungen für den Nutzer

Um den Nutzer optimal bei der Erstellung und Formulierung von Prompts zu unterstützen, wurden in die grafische Benutzeroberfläche des Moduls zwei Hilfsfenster integriert.

Das erste Hilfsfenster stellt Informationen bereit, welche im Rahmen des Evaluationskapitel noch erhoben werden sollen.

Ein zweites Hilfsfenster unterstützt den Nutzer bei der Auswahl relevanter Entitätstypen aus dem Domänenmodell, die in den Kontext der SQL-Generierung einfließen sollen. Dieses Fenster enthält erläuternde Texte und weist den Nutzer insbesondere darauf hin, auch verbindende BDOs einzubeziehen. Zusätzlich wird dem Nutzer eine automatisch generierte, auf UML basierende, visuelle Darstellung der aktuellen Beziehungsgraphen des Domänenmodells präsentiert. Die Visualisierung wird im wesentlichen von PlantUML übernommen, einer Bibliothek zur Generierung von UML-Diagrammen aus strukturierten, textuellen Beschreibungen von Datenmodellen.[9] Die Generierung und Einbindung erfolgt asynchron, um eine flüssige Bedienbarkeit zu gewährleisten.

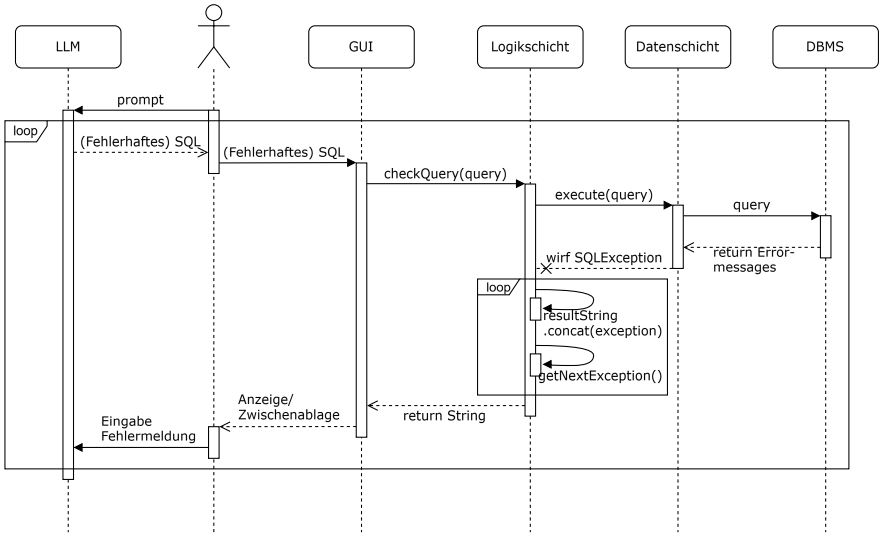

Abb. 5: Sequenzdiagramm: Fehlermeldungen aus `SQLExceptions` auslesen.

Schließlich ermöglicht das Tool dem Nutzer, die generierte SQL-Abfrage direkt in der GUI testweise auszuführen. Hierzu wird ein separates Fenster geöffnet. Das Ergebnis der Testausführung wird unmittelbar angezeigt: Bei Erfolg erhält der Nutzer eine positive Rückmeldung, andernfalls werden die Fehlermeldungen des Datenbanksystems, extrahiert aus der `SQLException` ausgegeben. Eine Eigenheit von `SQLException`s ist, dass diese verkettet vorliegen, falls mehrere Fehler aufgetreten sind.[10] So wird in der präsentierten Implementierung so lange durch die Exceptions mittels `.getNextException()` durchiteriert, bis schließlich `null` zurückgegeben wird und somit das Ende der Kette erreicht wurde. Aus jeder so betrachteten Exception wird die genaue Fehlerbeschreibung extrahiert und diese gesammelt zurückgegeben. Der Mechanismus ist im Sequenzdiagramm in Abb. 5 dargestellt. Die gesammelten Fehlermeldungen werden präsentiert und können direkt in die Zwischenablage kopiert und anschließend als Feedback an das verwendete LLM weitergegeben werden, wodurch die generierten SQL-Abfragen iterativ verbessert werden können.

6 Evaluation

Im Rahmen der Evaluation wird untersucht, wie gut der Ansatz für verschiedene Vorgehensweisen funktioniert. Final wird aus den Ergebnissen eine Handlungsempfehlung für die Nutzer der eHÜL abgeleitet, die im bereits implementierten Hilfsfenster zur Verfügung gestellt wird.

6.1 Einflussfaktoren auf die Generierung

Die Generierung von SQL-Statements durch LLMs wird von mehreren Faktoren beeinflusst. Im Folgenden werden drei zentrale Fragen untersucht:

- **Welchen Einfluss hat die Wahl des konkreten LLMs auf das generierte SQL-Statement?** Welche Modelle liefern präzisere und korrektere Ergebnisse?
- **Welchen Einfluss hat die Formulierung des Nutzerprompts auf das Resultat?** Wie unterscheiden sich die generierten SQL-Statements je nach gewähltem Formulierungsstil?
- **Wie wirkt sich die Komplexität der geforderten Queries auf das Ergebnis aus?** Gibt es eine Grenze, ab der LLMs fehleranfälliger werden?

Diese Fragestellungen werden im Folgenden systematisch untersucht, indem bestehende SQL-Abfragen aus dem Produktivsystem analysiert, verschiedene Formulierungsarten angewendet und mehrere Modelle getestet werden.

Modellwahl

Eine Vielzahl an LLMs steht für die Generierung von SQL-Abfragen zur Verfügung. Ziel ist, zu evaluieren, welche Modelle sich am besten eignen. Dabei werden folgende bekannte Systeme getestet:

- **ChatGPT** (OpenAI)
- **Gemini** (Google DeepMind)
- **Claude** (Anthropic)
- **DeepSeek** (DeepSeek AI)

Bei ChatGPT, Gemini und Claude handelt es sich um drei marktführende Sprachmodelle aus dem kommerziellen Sektor.[11] DeepSeek ist ein Open-Source-Modell und lässt sich somit auch lokal ausführen[12], was es für die Anwendung im Institut für Informatik der Universität Bonn interessant macht.

Jedes Modell erhält identische Eingaben, sodass Unterschiede in der Generierung direkt verglichen werden können.

Formulierungsarten von Nutzerprompts

Die Art und Weise, wie Nutzer ihre Abfragen formulieren, beeinflusst die Qualität der von LLMs generierten SQL-Statements.[13] Zur systematischen Evaluierung wurden folgende vier Formulierungsarten entworfen:

Deskriptive Erklärung: Bei dieser Formulierung wird die gewünschte Abfrage in natürlich-sachlicher Sprache so beschrieben, wie ein Nutzer sie informell erklären würde. Sie enthält meist vollständige Sätze und erläutert Bedingungen und Filter in Klartext, ohne technische Kürzel oder Tabellennamen. Ziel ist eine zugängliche, leicht verständliche Form, die auch ohne Datenbankkenntnisse formulierbar ist.

Technische Formulierung: Hier wird die Abfrage mit präzisen, domänenspezifischen Begriffen beschrieben. Tabellennamen, Attributnamen und konkrete Filterbedingungen werden explizit genannt.

Zielbezogene Promptformulierung: Diese Variante beschreibt ausschließlich das gewünschte Ergebnis, ohne Vorgaben zur konkreten Umsetzung in SQL. Nutzer formulieren dabei eher eine Frage oder ein Ziel.

Kurzform/Schlüsselwort-Prompt: Diese Formulierungsart besteht aus sehr knappen Eingaben in Form von Stichworten oder Listen relevanter BDOs und Attribute. Sie liefert minimale Information.

Für jede dieser Formulierungsarten wurde pro SQL-Abfrage ein passender Prompt entwickelt. Diese sind an entsprechender Stelle im Anhang zu finden.

Komplexität der SQL-Abfragen

Die im eHÜL-Produktivsystem gespeicherten SQL-Abfragen dienen als Referenz, um die Fähigkeit der Modelle zur Generierung unterschiedlich komplexer Abfragen

Anhang-Pos.	Name
Einfach	Wenige `JOIN`s, einfache Bedingungen. Es werden keine Subqueries oder Aggregationen verwendet.
Mittel	Mehrere `JOIN`s, einfache Aggregationen, einfache Subqueries.
Komplex	Mehrere `JOIN`s, komplizierte Subqueries, komplexe Filterlogik, Aggregationen über mehrere Tabellen.

Tabelle 1: Die ordinalen Komplexitätsstufen, zwecks Kategorisierung von SQL-Abfragen.

zu evaluieren. Hierbei werden drei Komplexitätsstufen unterschieden. Diese sind in der Tabelle 1 aufgeführt.
Folgend werden die im Produktivsystem verwendeten Abfragen anhand der Kriterien, die sie erfüllen, in diese Stufen eingeteilt. Diese Zuteilung ist in der Tabelle 2 dargestellt.

Anhang-Pos.	Name	Komplexität
A.1	Buchungen bestimmter Finanzierungen	Mittel
A.2	Sachbuchungen mit Rechnungs- und PSP-Daten	Einfach
A.3	Finanzierungen mit vollständigen Daten	Mittel
A.4	Kostengruppen mit zugehörigen PSP-Elementen	Einfach
A.5	Personalmittel bestimmter Personen	Komplex
A.6	Personalmittelfinanzierungen	Komplex
A.7	Betragssummen für Sachbuchungen	Mittel
A.8	Constraint: Budget der Finanzierungen	Einfach
A.9	Problembuchungen	Komplex

Tabelle 2: Die Einteilung der SQL-Abfragen im Anhang in die zuvor definierten Komplexitätsstufen.

Die drei Fragestellungen (Modellwahl, Formulierung des Prompts und Komplexität) werden gemeinsam betrachtet. Dadurch kann für jedes Modell ermittelt werden, welche Formulierungsart bis zu welcher Komplexitätsstufe wie gut funktioniert. Dies ermöglicht eine differenzierte Bewertung der Ansätze.

6.2 Auswertung

Untersucht werden $n_{llm} = 4$ Sprachmodelle, mit $n_{pfa} = 4$ Promptformulierungsarten von $n_{ma} = 9$ Muster-Abfragen. Um Zufälle auszuglätten, wird für jeden Prompt und jedes Modell die Generierung $n_{rep} = 5$-mal wiederholt. Die

Gesamtanzahl der durchzuführenden LLM-Abfragen n_{total} berechnet sich also nach

$$n_{llm} \cdot n_{pfa} \cdot n_{ma} \cdot n_{rep} = n_{total}$$

und evaluiert mit den genannten Werten zu 720.

Dies ist offensichtlich eine zu hohe Zahl, um dies händisch durchzuführen. Entsprechend wird ein eigens entwickeltes Tool genutzt, das diese Aufgabe übernimmt, indem es mit den APIs der jeweiligen LLMs kommuniziert und die Ergebnisse in auswertbarer Form abspeichert.[1]

Das zugehörige interne Datenmodell des Auswertungsprogramms ist in Abb. 6 dargestellt. Manuell einzutragen sind die hier schon genannten Muster-Abfragen aus dem Produktivsystem, die Promptformulierungsarten, die LLMs, sowie für jedes Paar aus Muster-Abfrage und Promptformulierungsart ein entsprechender Prompt. Diese Prompts finden sich im Anhang unter den zugehörigen Abfragen.

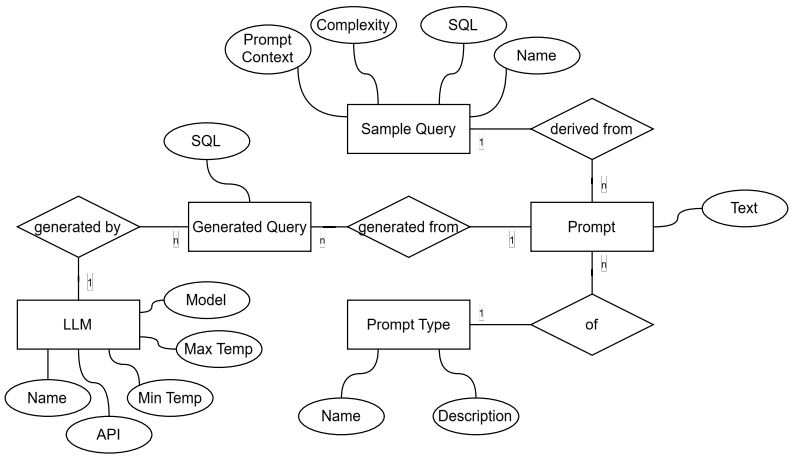

Abb. 6: ER-Diagramm: Das Datenmodell des Auswertungstools. Der Nutzer trägt zunächst alle Muster-Abfragen (Sample Query) aus dem Produktivsystem ein, danach die LLMs die genutzt werden sollen, sowie die Formulierungsarten (Prompt Type). Für jedes Paar aus Formulierungsart und Muster-Abfrage wird dann ein Prompt entworfen. Das Programm generiert dann auf Basis dieser Prompts mit den LLMs die generierten Abfragen (Generated Query).

[1] Das zur Generierung und Auswertung verwendete Tool ist öffentlich unter `https://github.com/felixsegg/sql-analyzer` verfügbar.

Im ER-Diagramm fällt auf, dass für die LLMs eine Minimal- und Maximaltemperatur gesetzt werden kann. Das setzen dieser zwei Werte, anstatt nur eines einzigen, hat folgenden Hintergrund: Viele LLMs nutzen Caching. Dabei wird für zuvor gestellte Prompts die Antwort gespeichert und im Falle einer erneuten Abfragestellung mit selbem Inhalt die gleiche Antwort zurückgegeben. Die Gründe hierfür sind einleuchtend: Das Vorgehen spart Ressourcen und führt natürlich auch zu einer kürzeren Antwortzeit, da das LLM selber nicht aktiv werden muss, sondern schlicht die vorige Antwort zurückgegeben werden kann.

In dem hier betrachteten Anwendungsfall ist das Caching jedoch unerwünscht, da gerade die Mannigfaltigkeit der erhaltenen Antworten untersucht werden soll. Entsprechend wird für jede aus dem Auswertungsprogramm ausgehende Abfrage an die LLMs eine individuelle Temperatur gewählt, in gleichförmigen Schritten von der eingestellten Minimal- zur eingestellten Maximaltemperatur. Dies verhindert effektiv das Caching. Anekdotische Nutzerberichte deuten darauf hin, dass die Temperature-Einstellung in vielen LLM-Weboberflächen um 0,7 liegt.[14] Deswegen wurden auch hier eine Temperaturintervall von 0,7 bis 0,8 gewählt; jeder Prompt an die APIs wird somit mit einer Temperatur aus dem folgenden Pool gestellt:

$$\{0.7, 0.725, 0.75, 0.775, 0.8\}$$

Die konkret über die jeweiligen APIs angesprochenen Modelle sind in Tabelle 3 angegeben. Die Auswahl erfolgte auf Basis von Abwägungen zwischen Aktualität, Verfügbarkeit der Modelle für Nutzer in Weboberflächen und Preis-Leistungs-Überlegungen in Bezug auf die jeweiligen APIs.

LLM	Modell	API-Modell-Schlüssel
ChatGPT	GPT-4o mini	`gpt-4o-mini-2024-07-18`
Google Gemini	Gemini 2.5 Flash (Preview)	`gemini-2.5-flash-preview-05-20`
Anthropic Claude	Claude Sonnet 4	`claude-sonnet-4-20250514`
DeepSeek AI	DeepSeek R1	`deepseek-reasoner`

Tabelle 3: Die gewählten KI-Modelle. Der Abruf erfolgte am 14. Juni 2025.

Nachdem das Auswertungsprogramm die generierten Abfragen entgegengenommen und abgespeichert hat, wertet es diese aus, indem es die generierten Statements mit der Muster-Abfrage vergleicht. Für diesen Schritt sind mehrere Ansätze denkbar, im folgenden werden drei vorgestellt und verglichen: ein syntaktisch orientierter Vergleich, ein datenbasierter semantischer Vergleich

sowie ein Vergleich durch ein weiteres Sprachmodell.

Ein erster, naheliegender Ansatz besteht im syntaktischen Vergleich. Hierbei wird das SQL-Statement analysiert, in seine Bestandteile zerlegt und mit der Struktur der Musterabfrage verglichen. Für jeden der typischen Bestandteile (`SELECT, WHERE, FROM, ORDER BY,` ...) wird eine Ähnlichkeitsbewertung durchgeführt, etwa mittels des Jaccard-Index zum Mengenvergleich, und anschließend gewichtet zu einem Gesamtwert aggregiert. Trotz der Tatsache, dass dieser Ansatz strukturierte und reproduzierbare Resultate liefert, lässt er keine zufriedenstellende Bewertung semantischer Äquivalenz zu. SQL ist eine deklarative Abfragesprache, so lassen sich zwei äquivalente Abfragen auf unterschiedlichste Weise korrekt auszudrücken, in etwa durch die Umstrukturierung von Bedingungen oder der Nutzung von Subqueries. Der syntaktische Vergleich liefert somit bei funktional identischen, aber formal abweichenden Abfragen unbefriedigende Ergebnisse.

Ein zweiter Ansatz zielt darauf ab, die tatsächliche Semantik der Abfragen über deren Ausführung zu bewerten. Hierbei würden beide Statements auf derselben Datenbasis ausgeführt und die dabei entstehenden Resultatmengen verglichen. In der Theorie verspricht dieser Weg eine hohe Genauigkeit, da nur das tatsächliche Verhalten der Abfragen zählt. In der Praxis erweist sich dieser Ansatz jedoch als nur sehr schwer umsetzbar. Die Testdaten müssten einerseits realitätsnah, andererseits ausreichend vollständig und vielfältig sein, um die Wirkung der Abfragen adäquat abbilden zu können. Viele Abfragen beinhalten jedoch Parametrisierungen oder Bedingungen, deren Ergebnis stark von konkreten Werten abhängt. Zudem können selbst sehr unterschiedliche Abfragen identische oder leere Resultsets liefern.

Ein dritter Ansatz nutzt wiederum ein Sprachmodell, diesmal jedoch nicht zur Generierung, sondern zur Bewertung. Dabei wird dem Modell ein Prompt mit dem Muster-Statement (dabei handelt es sich um die aus dem Produktivsystem genommenen Abfragen, die auch im Anhang zu finden sind) und dem generierten Statement übergeben. Dieser fordert das Modell auf, die beiden Abfragen hinsichtlich ihrer semantischen Ähnlichkeit zu bewerten und dabei eine Zahl zwischen 0 (keinerlei Übereinstimmung) und 100 (semantisch äquivalent) zurückzugeben. Die Ergebnisse werden anschließend zur weiteren Analyse normiert. Um die Bewertung möglichst reproduzierbar zu halten, wird das Sampling über die Modellparameter der API deaktiviert (`temperature = 0`).

Dieser LLM-gestützte Bewertungsansatz bringt offensichtlich gewisse Nachteile mit sich. Die Bewertung ist nicht vollständig deterministisch, da auch bei deaktiviertem Sampling gelegentlich kleinere Schwankungen auftreten können und bei Wiederholung ein neueres Modell auch abweichende und somit nicht reproduzierende Ergebnisse liefert. Zudem ist der Ansatz durch die Modellantwort in gewissem Maß subjektiv. Nichtsdestotrotz überwiegen die praktischen Vorteile: Sprachmodelle sind in der Lage, auch bei stark abweichender Syntax semantische

Gemeinsamkeiten zu erkennen, die algorithmisch schwer zu fassen wären. Gerade in einem System, das selbst auf Sprachmodellen basiert, ist diese Form der Evaluierung besonders naheliegend. Die Reproduzierbarkeit ist bei gleichbleibenden Prompts und deaktiviertem Sampling ausreichend gegeben. Dementsprechend wird an dieser Stelle der LLM-basierte Vergleich gewählt.

Aus Praktikabilitätsgründen wurde zur Bewertung Google Gemini gewählt, wie es in der Tabelle 3 zu finden ist.

Das folgende Bewertungsschema wird dem LLM mit übergeben:

- **0–5:** Kein oder kaum erkennbarer Zusammenhang. Eine Umformulierung wäre aufwendiger als ein kompletter Neustart.
- **6–25:** Semantisch extrem unterschiedlich, nur sehr schwache Ähnlichkeit erkennbar.
- **26–45:** Semantisch deutlich verschieden, aber eine grobe thematische Nähe ist vorhanden.
- **46–60:** Semantisch nicht äquivalent, aber mit klar erkennbarer gemeinsamer Grundlage. Überarbeitung wäre mit etwas Erfahrung oder KI-Hilfe gut machbar.
- **61–85:** Semantisch nicht exakt äquivalent, aber der Unterschied ist gering und leicht korrigierbar.
- **86–99:** Semantisch fast äquivalent, Unterschiede nur in minimalen Details.
- **100:** Semantisch vollständig äquivalent; Unterschiede höchstens bei Spaltenauswahl oder -reihenfolge.

Der folgende Prompt wurde verwendet, um die Bewertung der Prompt-Paare von Google Gemini generieren zu lassen.

```
Du bekommst zwei SQL-Select-Statements. Das erste ist eine
Musterlösung. Das zweite wurde anhand einer informellen
Beschreibung des Ziels des ersten Statements nachempfunden.
Vergleiche beide Statements hinsichtlich ihrer semantischen
Ähnlichkeit. Aliase sind irrelevant. Entscheidend ist
ausschließlich, ob ein semantisch äquivalenter Weg gewählt wurde.
Die konkrete Syntax spielt nur eine untergeordnete Rolle.
Wichtig: Gib ausschließlich eine ganze Zahl zwischen 0 und 100
zurück - keinerlei weitere Zeichen als Erklärung, auch ohne
zusätzliche Zeichen oder Formatierungen.
Vermeide Rundungen auf Vielfache von 5, außer sie sind sachlich
gerechtfertigt. Nutze feine Abstufungen in Einerschritten.
Wähle immer die Zahl, die die tatsächliche semantische Nähe am
präzisesten widerspiegelt. Vermeide z.B. 60, 75 oder 85, wenn 62,
76 oder 84 genauer wären.
Scheue nicht davor zurück, die vollen 100 Punkte zu vergeben, wenn
eine semantische Äquivalenz vorliegt.
```

```
Bewerte nach folgendem Raster:
- 0-5: Kein oder kaum erkennbarer Zusammenhang. Eine Umformulierung
wäre aufwendiger als ein kompletter Neustart.
- 6-25: Semantisch extrem unterschiedlich, nur sehr schwache
Ähnlichkeit erkennbar.
- 26-45: Semantisch deutlich verschieden, aber eine grobe thematische
Nähe ist vorhanden.
- 46-60: Semantisch nicht äquivalent, aber mit klar erkennbarer
gemeinsamer Grundlage. Überarbeitung wäre mit etwas Erfahrung oder
KI-Hilfe gut machbar.
- 61-85: Semantisch nicht exakt äquivalent, aber der Unterschied ist
gering und leicht korrigierbar.
- 86-99: Semantisch fast äquivalent, Unterschiede nur in minimalen
Details.
- 100: Semantisch vollständig äquivalent; Unterschiede höchstens bei
Spaltenauswahl oder -reihenfolge.

Muster-Statement:
(
...
)

Nachempfundenes Statement:
(
...
)
```

Die Ergebnisse unterliegen unter diesem Ansatz so natürlich auch einigen Limitierungen. Die semantische Bewertung der Resultate bleibt abhängig von einem weiteren Sprachmodell und somit subjektiv. Die konkreten Formulierungsarten werden sich in der Praxis je nach Nutzer oder sogar je nach Situation und Zeitpunkt unterscheiden können. Entsprechend ist die Übertragbarkeit und Verallgemeinerung der Ergebnisse mit Vorsicht zu betrachten. Schließlich wurden die Modelle nur auf einem Ausschnitt typischer Abfragen getestet, sodass Übertragbarkeit auf alle möglichen SQL-Abfragen im System nicht garantiert ist.

6.3 Ergebnisse

Eine Generierung und Auswertung von LLM-generierten SQL-Statements wurde nach dem soeben vorgestellten Prinzip vorgenommen.[2] Im Folgenden werden die Ergebnisse nach den drei untersuchten Einflussfaktoren Modell, Promptformulierung und Abfragekomplexität analysiert. Die genauen R-Skripte,

[2] Die csv-Datei mit den erhobenen Werten wurde gemeinsam mit dieser Arbeit beim Prüfungsamt eingereicht.

die zur grafischen Aufbereitung der Ergebnisse verwendet wurden, finden sich im Anhang B.

In Abb. 7 ist der Einfluss der untersuchten Sprachmodelle auf die Bewertung dargestellt. Diese zeigt, dass Claude (Ø 0,82) die beste Gesamtleistung erzielt. Auch DeepSeek (Ø 0,81) liegt auf hohem Niveau, bei zugleich geringster Streuung ($\sigma \approx 0,12$), was auf eine konsistent starke Performance hindeutet. Gemini (Ø 0,79) folgt mit geringem Abstand, zeigt sich robust, ohne dabei Spitzenwerte zu erreichen. ChatGPT (Ø 0,66) bleibt mit Abstand zurück und weist zudem die höchste Streuung auf ($\sigma \approx 0,22$), was auf eine stark schwankende Qualität der Ausgaben schließen lässt.

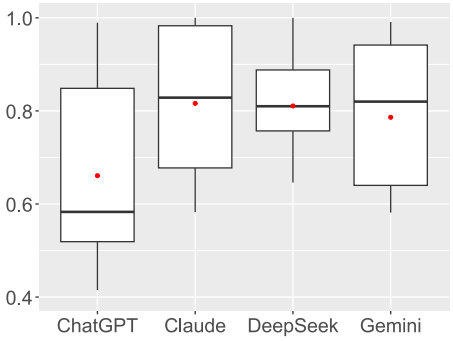

Abb. 7: Boxplot der durchschnittlichen Scores für jede Abfrage nach verwendetem LLM. Claude und DeepSeek erzielen die höchsten Durchschnittswerte, ChatGPT zeigt die größte Streuung. Der Punkt zeigt den Durchschnitt, der Strich den Median.[15][16][17]

Der Einfluss der Abfragekomplexität ist in Abb. 8 dargestellt. Die Analyse ergibt ein überraschendes Bild: Einfache Abfragen (Ø 0,82) erzielen erwartungsgemäß die besten Ergebnisse. Mittlere Komplexität (Ø 0,73) führt jedoch zu schlechteren Scores als hohe Komplexität (Ø 0,76). Betrachtet man den Median sind die Differenzen noch extremer. Die stärkste Streuung tritt ebenfalls bei mittlerer Komplexität auf ($\sigma \approx 0,21$).

Der Einfluss der Promptformulierung (Abb. 9) zeichnet folgendes Bild: Die technische Formulierung ist mit Abstand am effektivsten (Ø 0,82) und liefert häufig sehr hohe Scores (Q3 = 1,00). Dagegen schneidet die Kurzform (Ø 0,74) schlechter ab. Die zielbezogene Variante liegt im Mittelfeld (Ø 0,75), genau wie deskriptive Prompts (Ø 0,75). Die deskriptive Variante zeichnet im Median

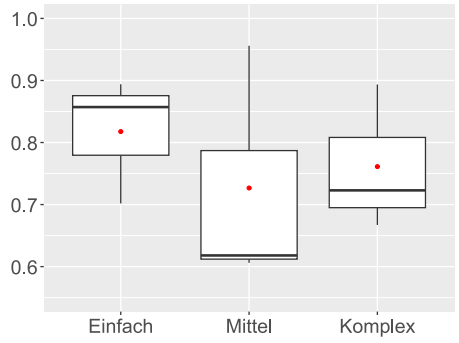

Abb. 8: Boxplot der Scores gruppiert nach Komplexitätsstufen der Musterabfragen. Abfragen mittlerer Komplexität schneiden signifikant schlechter ab als einfache oder komplexe. Der Punkt zeigt den Durchschnitt, der Strich den Median.[15][16][17]

jedoch ein schlechtes Bild. Hier bestätigt sich die Intuition: Je präziser und technischer der Prompt, desto besser das Ergebnis.

Eine differenzierte Betrachtung nach Modellwahl und Abfragekomplexität ist in Abb. 10 dargestellt. Diese zeigt, dass Claude besonders bei einfachen (0,90) und hohen (0,83) Komplexitäten stark ist. DeepSeek erzielt den höchsten Einzelwert überhaupt mit 0,91 bei hoher Komplexität – ein überraschendes Ergebnis. ChatGPT hingegen fällt bei hoher Komplexität dramatisch ab (0,48).

Weiter wird in Abb. 11 die Interaktion von Formulierungsarten und Komplexitäten der geforderten Statements dargestellt. Technische Prompts schneiden bei einfachen (0,92) und komplexen (0,84) Abfragen sehr gut ab. Wieder überraschend aber passend zu den bisherigen Ergebnissen verlieren sie bei mittlerer Komplexität an Wirksamkeit (0,71). Erneut zeigt sich eine klare Präferenz der Modelle für explizite, strukturierte Eingaben.

Die dritte Heatmap (Abb. 12) aus den Variablen des Sprachmodells und der Promptformulierung bestätigt die bisherigen Befunde: Claude und DeepSeek profitieren stark von technisch präzisen Prompts (jeweils 0,90). Gemini zeigt über alle Promptformen hinweg eine solide Leistung, während ChatGPT bei deskriptiven Prompts besonders schwach ist (0,63).

Das facettierte Säulendiagramm (Abb. 13) vereint alle drei Einflussfaktoren und macht deutlich, dass technische Promptformulierungen bei fast allen Modellen und Komplexitätsstufen die höchsten Mittelwerte erzielen. Zugleich zeigt sich erneut, dass Claude und DeepSeek auch bei hoher Komplexität robuste Leistungen

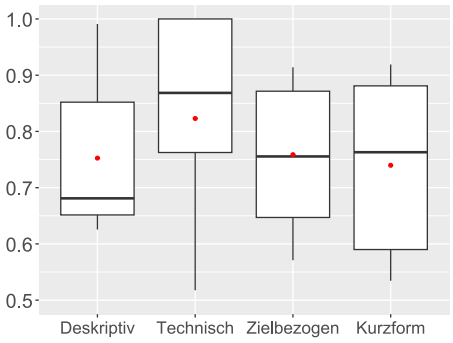

Abb. 9: Boxplot der Scores in Abhängigkeit von der gewählten Promptform. Technische Prompts liefern die höchsten und stabilsten Ergebnisse. Der Punkt zeigt den Durchschnitt, der Strich den Median.[15][16][17]

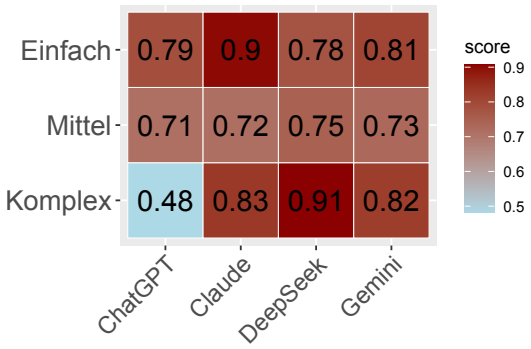

Abb. 10: Heatmap: Durchschnittlicher Score je Modell und Abfragekomplexität. DeepSeek übertrifft alle anderen bei komplexen Abfragen; ChatGPT fällt dort deutlich ab.[15][16][17]

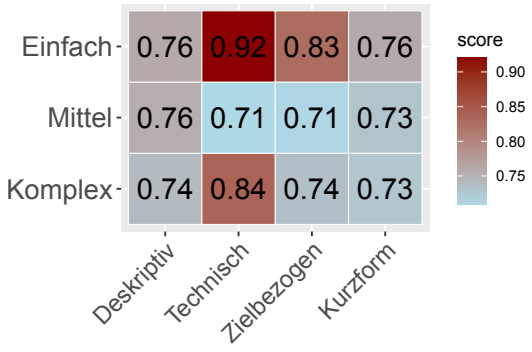

Abb. 11: Heatmap: Durchschnittlicher Score nach Prompttyp und Abfragekomplexität. Technische Prompts sind besonders bei einfachen und komplexen Aufgaben überlegen.[15][16][17]

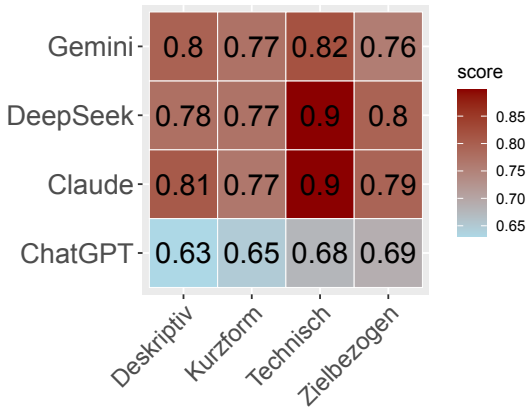

Abb. 12: Heatmap: Durchschnittlicher Score je Kombination aus LLM und Promptform. Claude und DeepSeek profitieren besonders stark von technisch präzisen Formulierungen.[15][16][17]

erbringen. ChatGPT hingegen scheint bei einfachen Abfragen noch Anschluss zu finden, bei steigender Komplexität fällt es jedoch deutlich zurück. Mittlere Komplexität bleibt insgesamt die forderndste Stufe, da hier bis auf die Ausnahme von ChatGPT die Balken aller Modelle und Prompttypen am niedrigsten ausfallen. Diese Zusammenführung bestätigt somit die Einzelergebnisse der Boxplots und Heatmaps: Für konsistente und möglichst präzise SQL-Generierung ist die Wahl eines leistungsstarken Modells (vorzugsweise Claude oder DeepSeek) in Kombination mit einer technisch-exakten Promptformulierung entscheidend.

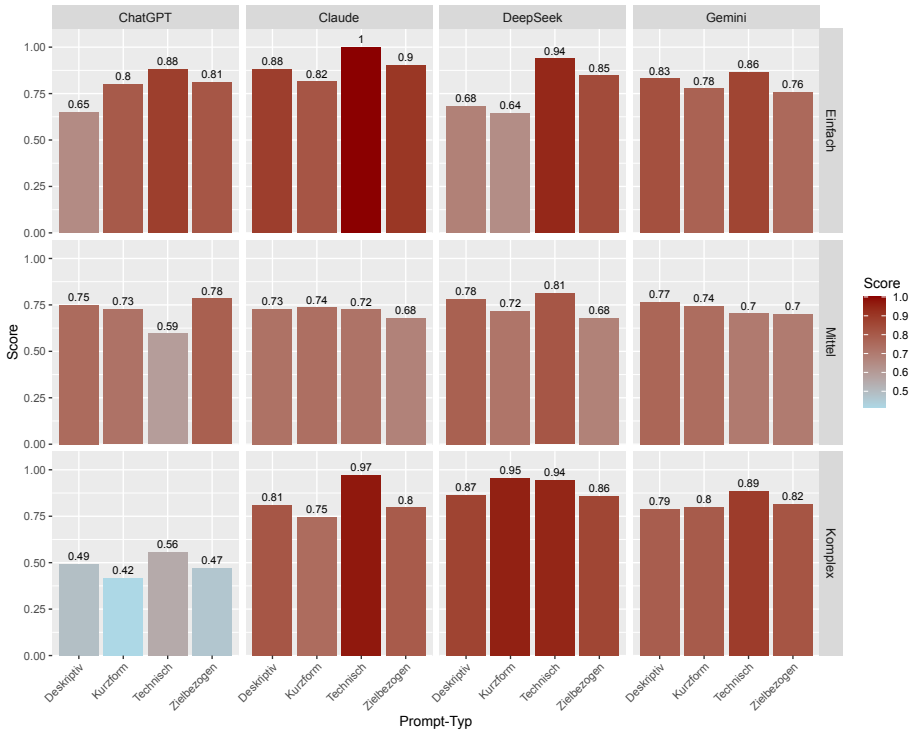

Abb. 13: Facettiertes Säulendiagramm: Durchschnittlicher Score für jede Kombination aus LLM, Promptformulierung und Abfragekomplexität.[15][16][17]

Zusammenfassend zeigt sich, dass die Wahl des Sprachmodells sowie die Formulierung des Nutzerprompts entscheidend für die Qualität der generierten SQL-Abfragen sind. Claude und DeepSeek erzielten insgesamt die zuverlässigsten Ergebnisse. ChatGPT blieb deutlich hinter den Erwartungen zurück, besonders bei hoher Komplexität zeigte es erhebliche Schwächen.

Überraschenderweise stellte sich heraus, dass nicht die komplexesten, sondern die Abfragen mittlerer Komplexität am schwierigsten zu generieren waren. Eine plausible Erklärung hierfür könnte die unterschiedliche Präzision der Prompts je nach Komplexität sein: Komplexe Abfragen motivieren häufig zu detaillierteren und sorgfältigeren Prompts, während mittlere Komplexitätsstufen möglicherweise oberflächlicher formuliert werden, wodurch sie stärker von Mehrdeutigkeiten betroffen sein könnten.

Die technische Formulierung der Prompts erwies sich insgesamt als die wirksamste Methode, da sie die präzisesten Ergebnisse und die geringste Streuung lieferte.

Auf Basis dieser Ergebnisse können den Nutzern des Generierungstools innerhalb der eHÜL wertvolle Hinweise bezüglich der Auswahl des Modells und der Formulierung der Prompts gegeben werden.

7 Ausblick und Fazit

7.1 Ausblick

Nach Entwurf und Fertigstellung des Tools bietet es weiterhin Raum für zukünftige Erweiterungen.

Eine zunächst festzustellende Stärke der Implementierung ist sein dynamischer Ansatz. Alle benötigten Informationen für die Kontextualisierung des Prompts mit Informationen über das Domänen- und Datenbankmodell werden zur Laufzeit gewonnen. Somit ist bei einer Weiterentwicklung dieser Modelle keine Anpassung des hier vorgestellten Programmcodes notwendig. Lediglich bei einer Abkehr von Hibernate (oder der JPA an sich) müsste der Ansatz geändert werden.

Ebenfalls positiv hervorzuheben ist die Tatsache, dass die genutzten LLMs mit der Zeit immer leistungsfähiger werden. Somit ist davon auszugehen, dass sich die Qualität der generierten SQL-Ausdrücke ohne Zutun der eHÜL-Entwickler in der Zukunft weiter verbessern wird.

Entwicklungsbedarf liegt noch in der Untersuchung der einzelnen Sprachmodelle. So wurde für jede der genannten LLMs lediglich eins von vielen Modellen repräsentativ für das Sammelsurium von Modellen der jeweiligen Entwicklerteams gewählt. Auch eine noch differenziertere Betrachtung der Promptformulierungsarten kann sich lohnen. So ließen sich noch weitere dieser Formulierungsvarianten definieren und überprüfen, oder für jedes Paar aus Formulierungsart und Muster-Abfrage verschiedene Prompts formulieren, um der Willkür der einzelnen im Rahmen dieser Arbeit entworfenen Prompts entgegenzuwirken. Weiter lässt sich die genaue Strukturierung der textuellen Beschreibungen des Domänen- und Datenbankmodells auf weitere Arten umsetzen. Ein letzter Punkt wäre die Untersuchung der Reaktion verschiedener Modelle auf die Konfrontation mit

der Fehlerausgabe des DBMS, wie sie im Rahmen der Fehlerüberprüfung der entworfenen Komponente dem Nutzer zur Rückgabe an das LLM präsentiert wird.

Programmatisch gibt es auch Potenzial für die Weiterentwicklung: So ließe sich ein Algorithmus entwerfen, der automatisch bei Wahl zweier Typen aus dem Domänenmodell einen „Beziehungsweg“ zwischen diesen beiden vorschlägt. Bei Bestätigung würden diese dann automatisch mit in die Menge der relevanten Typen für den Kontext mit aufgenommen.

Weiter sollte man auch diskutieren, ob wirklich die gesamte Bandbreite der Typen aus dem Domänenmodell vorgeschlagen werden soll. So liegen im Modell viele Klassen vor, die beispielsweise administrative Zwecke haben und nicht direkt aus dem Kontext der Finanzinformation stammen. Abb. 14 zeigt das von PlantUML generierte, recht unübersichtliche UML-Diagramm des BDMs, wie es dem Nutzer aktuell im Hilfefenster der Typenauswahl angezeigt wird. Auf der einen Seite liegt die Unübersichtlichkeit schlicht an der Komplexität des Datenmodells gepaart mit dem Umstand, dass es zur Laufzeit generiert wird. Zum anderen werden jedoch auch viele BDOs abgebildet, die für die meisten Nutzer möglicherweise gar nicht von Interesse sind. Ein Beispiel sind die eHÜL-Nutzer (in der Abbildung in einem separaten Teilgraphen auf der linken Seite zu sehen), für die die genauen Zugriffsrechte gespeichert werden. Es könnte sinnvoll sein, solche aus der Menge der für die Promptkontextualisierung auswählbaren Typen auszuschließen. Lediglich die verbleibende Teilmenge würde dem selektionsverwaltenden Objekt übergeben werden. Somit wäre dies leicht in der aktuellen Implementierung umzusetzen.

Letztlich sollte zukünftig auch das bereits existierende Parameterformat der dritten Stufe des QRL-Systems direkt vom LLM mit eingefügt werden können. Dies sollte sich leicht über eine entsprechende Erweiterung des Promptkontextes realisieren lassen.

7.2 Fazit

Im Rahmen dieser Arbeit wurde ein Mechanismus entwickelt und evaluiert, der es Nutzern ermöglicht, SQL-Abfragen intuitiv über das objektorientierte Domänenmodell des Finanzinformationssystems eHÜL zu formulieren. Ausgangspunkt war das Problem, für Nutzer mit ausschließlicher Erfahrung auf dem Domänenmodell SQL-Abfragen in der dritten Stufe des QRL-Systems über dem relationalen Schema zu formulieren. Zu dem Problem beigetragen hat der objektrelationale Mismatch, der durch strukturelle Diskrepanzen zwischen dem relationalen Datenbankschema und dem objektorientierten Domänenmodell entsteht.

Zur Lösung wurden zunächst verschiedene Ansätze diskutiert und schließlich ein KI-gestützter Ansatz gewählt. Dieser basiert im Wesentlichen auf der Nutzung

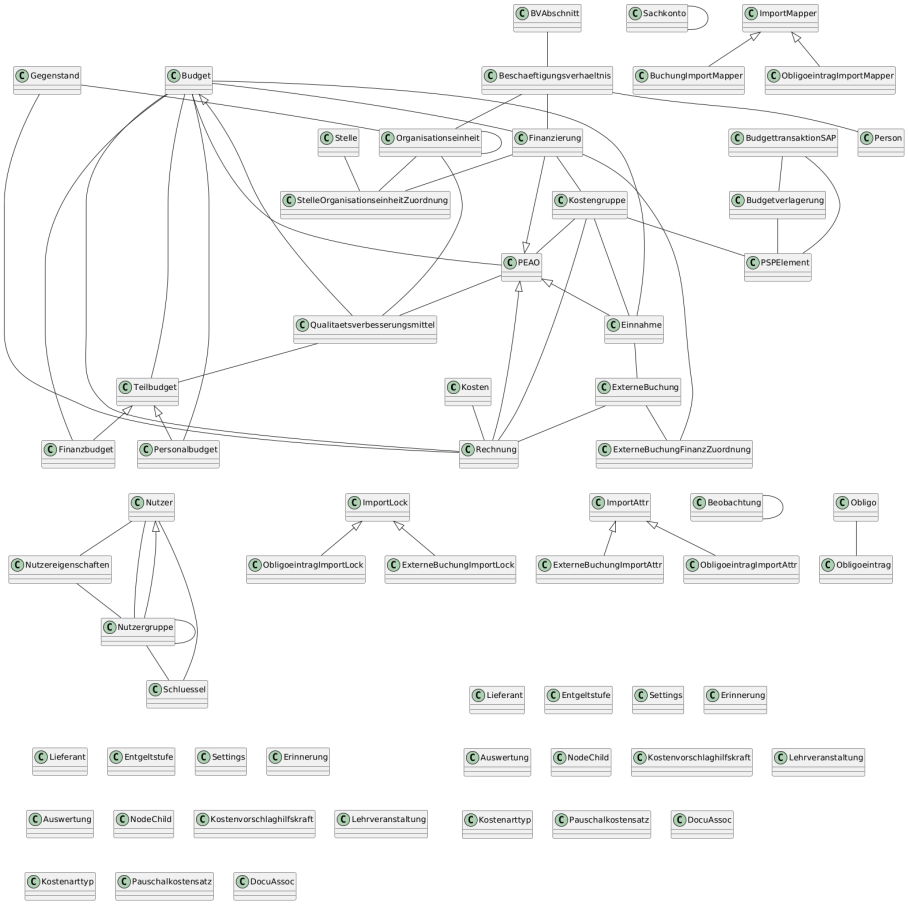

Abb. 14: Das UML-Diagramm des BDMs, wie es von PlantUML zur Laufzeit generiert und dem Nutzer im Programm aktuell als Hilfestellung angezeigt wird.

automatischer Generierung von SQL-Abfragen mittels LLMs. Nach Nutzereingabe eines geeigneten Prompts, formuliert über das Domänenmodell, wurde dieser Prompt mit Informationen über das Domänen- und Datenbankmodell sowie genauen Anweisungen für das Sprachmodell kontextualisiert. Zur Nutzung in der eHÜL wurde ein benutzerfreundliches Tool entwickelt. Dem Nutzer wird so eine intuitive grafische Oberfläche bereitgestellt. Mittels diesen Ansatzes wird das relationale Schema abstrahiert und Nutzern wird ermöglicht, auch komplexe SQL-Abfragen ohne tiefergehendes Wissen des zugrundeliegenden Datenbankschemas zu erstellen.

In der methodischen Evaluierung wurden mehrere populäre LLMs – Claude, DeepSeek, Gemini und ChatGPT – systematisch untersucht verglichen, wobei insbesondere die Wahl des Sprachmodells und die Präzision der Nutzerprompts maßgeblichen Einfluss auf die Qualität der generierten SQL-Abfragen hatten. Claude zeigte dabei die insgesamt zuverlässigsten Ergebnisse, auch DeepSeek ragte insbesondere bei komplexen Aufgaben heraus.

Das entwickelte Werkzeug erfüllt die initiale Zielsetzung, den Prozess der SQL-Generierung benutzerfreundlich, effizient und zuverlässig zu gestalten. Es eröffnet Nutzern neue Möglichkeiten, komplexe Abfragen einfacher und mit geringerem Aufwand zu erstellen.

Literatur

1. Bode, T. et al. *eHÜL Wiki*, Institut für Informatik, Universität Bonn. `https://gitlab.informatik.uni-bonn.de/projektgruppe-informationssysteme/ehuel/-/wikis/home`. Letzter Zugriff: 30. Juni 2025.
2. Jakarta Persistence Team. *Jakarta Persistence Specification*, Jakarta Persistence project. `https://jakartaee.github.io/persistence/latest/draft.pdf`. Letzter Zugriff: 30. Juni 2025.
3. Bauer, C., King, G. (2005). *Hibernate in action*, Manning Publications Co.
4. Ambler, S. W. (2003). *The object-relational impedance mismatch*, Agile Database Techniques, Wiley Publishing.
5. Liu, N. F. et al. (2023). *Lost in the Middle: How Language Models Use Long Contexts*, arXiv preprint arXiv:2307.03172.
6. Unterstein, M., Matthiessen, G., Unterstein, M., Matthiessen, G. (2013). *Der Systemkatalog.* Anwendungsentwicklung mit Datenbanken, 115-123.
7. Krishnamurthy, U. et al. (2025). *Oracle Database SQL Language Reference, 19c.* Oracle. `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/index.html`. Letzter Zugriff: 15. Juni 2025.
8. Windoffer, M. (2021). *Fortgeschrittene Abfragekonzepte für das datenbankgestützte Finanzinformationssystem eHÜL*, Bachelorarbeit Institut für Informatik, Universität Bonn.
9. PlantUML Team. (2025). *PlantUML Sprachreferenz (1.2025.0)*, . `https://plantuml.com/de/guide`. Letzter Zugriff: 26. Juni 2025.
10. Das, T. et al. (2025). *Oracle Database JDBC Developer's Guide, 21c*, Oracle. `https://docs.oracle.com/en/database/oracle/oracle-database/21/jjdbc/jdbc-developers-guide.pdf`. Letzter Zugriff: 17. Juni 2025.
11. Bailyn, E. (2025). *Generative AI Statistics: 2025 Report*, First Page Sage. `https://firstpagesage.com/seo-blog/generative-ai-statistics/`. Letzter Zugriff: 30. Mai 2025.
12. DeepSeek-AI. (2025). *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*, arXiv preprint arXiv:2501.12948. `https://arxiv.org/abs/2501.12948`. Letzter Zugriff: 21. Juni 2025.
13. Chang, S., Fosler-Lussier, E. (2023). *How to Prompt LLMs for Text-to-SQL: A Study in Zero-shot, Single-domain, and Cross-domain Settings.*, NeurIPS 2023 Table Representation Learning Workshop.
14. Ramlochan, S. (2024). *Complete Guide to Prompt Engineering with Temperature and Top-p*, Prompt engineering & AI Institute. `https://promptengineering.org/prompt-engineering-with-temperature-and-top-p/`. Letzter Zugriff: 30. Juni 2025.
15. R Core Team (2025). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. `https://www.R-project.org/`.
16. Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis.* Springer New York.
17. Wickham, H., François, R., Henry, L., Müller, K. (2023). *dplyr: A Grammar of Data Manipulation.* R package version 1.1.4.

A *Auswertungen* aus dem Produktivsystem

Im Folgenden sind einige Abfragen dargestellt, wie sie im eHÜL-Produktivsystem tatsächlich genutzt werden. An einigen Stellen wurden konkrete Werte zwecks Übersichtlichkeit weggekürzt und durch Auslassungspunkte ersetzt.

A.1 Buchungen bestimmter Finanzierungen

Die Abfrage liefert alle Buchungen, die mit einer Finanzierung einer bestimmten Unterkostenstelle verknüpft sind und im Jahr 2022 aktiv gewesen sind.

```
SELECT B.id, ...
FROM externebuchungsap B
WHERE B.id IN(
        SELECT EFZ.buchung_id
        FROM finanzierungsap F
            JOIN externebuchungfinanzsapzuordnung EFZ
                ON (F.id = EFZ.finanzierung_id)
        WHERE F.uks04 = 'InKind'
            AND '2022-07-01' <= F.bis
            AND '2022-12-31' >= F.von
    )
```

Formulierungsvarianten des Prompts

- **Deskriptiv:** Zeige alle Buchungen, die einer Finanzierung aus dem Jahr 2022 mit der 4. Unterkostenstelle 'InKind' zugeordnet sind.
- **Technisch:** Gib alle Datensätze aus 'externebuchungsap' zurück, deren id in der Tabelle externebuchungfinanzsapzuordnung' einer Finanzierung zugeordnet ist, bei der uks04 = 'InKind' gilt und deren Zeitraum in 2022 liegt.
- **Zielbezogen:** Welche Buchungen sind mit Finanzierungen aus 2022 der 4. Unterkostenstelle 'InKind' verknüpft?
- **Kurzform:** Buchungen in Finanzierungen, zweites Halbjahr 2022, vierte Unterkostenstelle 'InKind'

A.2 Sachbuchungen mit Rechnungs- und PSP-Daten

Die Abfrage gibt alle Sachbuchungen zurück, ergänzt um die zugehörigen Rechnungsinformationen und PSP-Elemente.

```
SELECT B.rechnungid, ...,
    R.status rstatus, ...,
    PSP.pspnr, ...
FROM externebuchungsap B
    LEFT JOIN rechnungsap R
        ON (R.id = B.rechnungid)
    LEFT JOIN tb.pspkg PSP
        ON (R.kgid = PSP.kgid)
WHERE B.psp = '...'
    AND B.buchtyp = 'Sachbuchung'
```

Formulierungsvarianten des Prompts

- **Deskriptiv:** Gib alle externe Buchungen vom Typ Sachbuchung des PSP-Elements '...' mit zugehörigen Informationen zu Rechnungen und Kostengruppen zurück,
- **Technisch:** Selektiere aus ExterneBuchung alle Einträge mit buchtyp = 'Sachbuchung' und psp = '...' inklusive der JOINs zu Rechnungen und Kostengruppe.
- **Zielbezogen:** Welche externe Buchungen vom Typ 'Sachbuchung' gehören zum PSP-Element '...' und wie lauten die zugehörigen Rechnungs- und Kostengruppen-Daten?
- **Kurzform:** Externe Buchung Typ 'sachbuchung', PSP-Element '...', Rechnung, Kostengruppe

A.3 Finanzierungen mit vollständigen Daten

Die Abfrage liefert alle Personen mit ihren Beschäftigungsverhältnissen, zugehörigen Finanzierungen und PSP-Elementen. Berücksichtigt werden nur Finan-

zierungen, die im zweiten Halbjahr 2022 aktiv waren und eine bestimmte vierte Unterkostenstelle haben. Die Ergebnisse sind nach Nachnamen sortiert.

```
SELECT P.id, P.personalnummer, P.nachname, P.vorname, P.aktenzeichen,
    BV.id, ...,
    PSP.pspid, ...,
    F.id, ...
FROM finanzierungsap F
    JOIN beschaeftigungsverhaeltnis BV
        ON (BV.id = F.bvid)
    JOIN person P
        ON (P.id = BV.personid)
    JOIN (
        SELECT PSP.id AS pspid, ...
            KG.id AS kgid, ...
            FROM kostengruppe KG
                JOIN pspelement PSP
                    ON (KG.pspid = PSP.id)
    ) PSP
        ON (F.kgid = PSP.kgid)
WHERE F.uks04 = '...'
    AND '2022-07-01' <= F.bis
    AND '2022-12-31' >= F.von
ORDER BY P.nachname
```

Formulierungsvarianten des Prompts

- **Deskriptiv:** Gib alle Personen mit ihren Beschäftigungsverhältnissen, Finanzierungen und PSP-Elementen aus, wenn die Finanzierungen im zweiten Halbjahr 2022 aktiv waren und die vierte Unterkostenstelle 'InKind' entspricht, geordnet nach Nachname.
- **Technisch:** Selektiere alle aus Finanzierung, Beschaeftigungsverhaeltnis, Person und PSPElement verknüpften Einträge, bei denen uks04 = 'InKind' und der Beginn der Finanzierung am oder vor '2022-07-01' und das Ende am oder nach '2022-12-31' liegt, geordnet nach Nachname der Person.
- **Zielbezogen:** Welche Personen waren über das gesamte 2. Halbjahr 2022 über Finanzierungen mit einer der vierten Unterkostenstelle 'InKind' aktiv beschäftigt? Inklusive der Daten zu Kostengruppe und PSP-Element und geordnet nach Nachname.
- **Kurzform:** Finanzierungen, Beschäftigungsverhältnisse, Personen, Kostengruppen, PSP-Elemente, zweites Halbjahr 2022, vierte Unterkostenstelle 'InKind', Sortierung nach Nachname

A.4 Kostengruppen mit zugehörigen PSP-Elementen

Die Abfrage gibt alle Kostengruppen aus und ordnet sie ihren jeweiligen PSP-Elementen zu.

```
SELECT PSP.id AS pspid, ...
    KG.id AS kgid, ...
FROM kostengruppe KG
    JOIN pspelement PSP
        ON KG.pspid = PSP.id
```

Formulierungsvarianten des Prompts

- **Deskriptiv:** Liste alle Kostengruppen mit den ihnen zugeordneten PSP-Elementen auf.
- **Technisch:** Zeige alle IDs sowohl aus Kostengruppe als auch aus PSPElement, gejoint miteinander.
- **Zielbezogen:** Welche IDs aus Kostengruppen und PSP-Elementen gehören zueinander?
- **Kurzform:** Kostengruppe- & PSP-Element-IDs, Zuordnung

A.5 Personalmittel bestimmter Personen

Die Abfrage gibt alle Finanzierungen aus, die sowohl mit mindestens einer der (im Array über ihre Personalnummer) angegebenen Personen über deren Beschäftigungsverhältnis verknüpft sind als auch einem Personalmittelbudget-Konto zugeordnet sind.

```
SELECT *
FROM Finanzierungsap F
WHERE EXISTS(
        SELECT 1
        FROM Person P JOIN Beschaeftigungsverhaeltnis BV
            ON (P.id = BV.personid)
        WHERE P.personalnummer IN (...)
            AND BV.id = F.bvid
    )
    AND EXISTS(
        SELECT 1
        FROM (
            SELECT *
            FROM kostengruppe KG
                JOIN pspelement PSP
                    ON KG.pspid = PSP.id
        ) PSPKG
        WHERE F.kgid = PSPKG.kgid
            AND PSPKG.pspbez = 'Personalmittelbudget'
    )
```

Formulierungsvarianten des Prompts

- **Deskriptiv:** Finde alle Finanzierungen, für die ein Beschäftigungsverhältnis einer Person mit Personalnummer '...', '...' oder '...' existiert und für die eine Kostengruppe mit PSP-Element mit Bezeichnung 'Personalmittelbudget' existiert.
- **Technisch:** Selektiere alle Einträge ausFinanzierung, die über die ID eines Beschäftigungsverhältnisses einer Person mit Personalnummer in ('0000', '1234', '6789') zugeordnet sind und deren PSP-Element die Bezeichnung 'Personalmittelbudget' trägt.
- **Zielbezogen:** Welche Finanzierungen haben Beschäftigungsverhätnisse von Personen mit Personalnummern '...', '...' oder '...' und beziehen sich auf ein PSP-Element mit Bezeichnung 'Personalmittelbudget'?
- **Kurzform:** Finanzierung mit Beschäftigungsverhältnis von Person mit Personalnummer aus '...', '...', '...' und PSP-Element-Bezeichnung 'Personalmittelbudget'

A.6 Personalmittelfinanzierungen

Die Abfrage liefert alle Finanzierungen zusammen mit den zugehörigen Beschäftigungsverhältnissen, Personen und PSP-Elementen, die:

- im Jahr 2023 aktiv sind,
- einer bestimmten Entgeltstufe zugeordnet sind,
- zu einem der angegebenen PSP-Elemente gehören,
- und für dieselbe Person mindestens eine weitere Finanzierung existiert, die zu einem bestimmten PSP-Element gehört.

```
SELECT P.id, ...,
    BV.id, ...,
    PSP.pspid, ...,
    F.id, ...
FROM finanzierungsap F
    JOIN beschaeftigungsverhaeltnis BV
        ON (BV.id = F.bvid)
    JOIN person P
        ON (P.id = BV.personid)
    JOIN tb.pspkg PSP
        ON (F.kgid = PSP.kgid)
WHERE f.von <= '2023-12-31'
    AND f.bis >= '2023-01-01'
    AND BV.entgeltstufe in (...)
    AND PSP.pspnr in (...)
    AND EXISTS(
        SELECT *
        FROM finanzierungsap Fi
            JOIN beschaeftigungsverhaeltnis BVi
                ON (BVi.id = Fi.bvid)
            JOIN tb.pspkg PSPi
                ON (Fi.kgid = PSPi.kgid)
        WHERE F.bvid = BVi.id
            AND PSPi.pspnr = '...'
    )
```

Formulierungsvarianten des Prompts

- **Deskriptiv:** Zeige alle im gesamten Jahr 2023 aktiven Finanzierungen mit zugehörigen Personen-, Beschäftigungsverhältnis- und PSP-Daten für Beschäftigungsverhältnis-Entgeltstufen ... und PSP-Kennungen mit Strich ..., sofern für dasselbe Beschäftigungsverhältnis eine weitere Finanzierung mit dem PSP-Element '...' existiert.
- **Technisch:** Abfrage auf Finanzierung, Beschaeftigungsverhaeltnis, Person, Kostengruppe und PSP-Element, mit Bedingungen auf Finanzierungsstart am oder vor dem '2023-01-01', Finanzierungsende am oder nach dem '2023-12-31', Beschäftigungsverhältnis-Entgeltstufen im Array (...), PSP-Kennungen mit Strich im Array (...), und Existenz einer weiteren Finanzierung des gleichen Beschäftigungsverhältnisses mit PSP-Kennung mit Strich '...'.
- **Zielbezogen:** Welche Personen haben welche über das gesamte Jahr 2023 aktive Finanzierungen mit Beschäftigungsverhältnis-Entgeltstufe ... und PSP-Element-Kennung mit Strich ... und zusätzlich eine zweite Finanzierung des Beschäftigungsverhältnisses mit PSP-Strichkennung '...'?
- **Kurzform:** Person, Beschäftigungsverhältnis mit Entgeltstufe ..., PSP-Element mit Strichkennung ..., Finanzierung durchgehend aktiv 2023, für selbes Beschäftigungsverhältnis weitere FInanzierung mit PSP-Element '...'

A.7 Betragssummen für Sachbuchungen

Die Abfrage summiert die Beträge aller Buchungen pro Rechnung, die einem bestimmten PSP-Element zugeordnet sind, Sachbuchungen sind und bereits als zugeordnet gelten.

```
SELECT B.rechnungid RID, SUM(B.betrag) bbetrag
FROM externebuchungsap B
WHERE B.psp = '...'
    AND B.buchtyp = 'Sachbuchung'
    AND B.status = 'zugeordnet'
GROUP BY B.rechnungid
```

Formulierungsvarianten des Prompts

- **Deskriptiv:** Ermittle die Summen aller Beträge gruppiert nach Rechnungs-ID für Buchungen vom Typ 'Sachbuchung' und Status 'zugeordnet' mit PSP-Element '...'.
- **Technisch:** Gruppiere ExterneBuchung nach Rechnungs-ID, filtere auf Buchungstyp = 'Sachbuchung', Status = 'zugeordnet', und PSP-Element = '...', und summiere den Betrag.
- **Zielbezogen:** Wie hoch sind die Gesamtbeträge je Rechnung für bestimmte Buchungen vom Typ 'Sachbuchung' und Status 'zugeordnet' mit PSP-Element '...'?
- **Kurzform:** ExterneBuchung, Betragssumme je Rechnung, Status 'zugeordnet', Buchungstyp 'Sachbuchung', PSP-Element '...'

A.8 Constraint: Budget der Finanzierung

Eine simple Abfrage, die genutzt wird, um bestimmte Constraints im System zu verfassen.

```
SELECT id
FROM finanzierungsap
WHERE abteilungpmb IS NOT NULL
    AND budgetpmb IS NULL
```

Formulierungsvarianten des Prompts

- **Deskriptiv:** Gib alle Finanzierungen aus, bei denen die Abteilung nicht fehlt, das Budget (PMB) aber schon.
- **Technisch:** Selektiere aus Finanzierung alle Datensätze mit Abteilung (PMB) ungleich NULL und Budget (PMB) gleich NULL.
- **Zielbezogen:** Welche Finanzierungen haben eine gesetzte Abteilung aber ein fehlendes Budget (PMB)?
- **Kurzform:** Finanzierung, Abteilung vorhanden, Budget (PMB) fehlt

A.9 Problembuchungen

Buchungen, die über unterschiedliche Finanzierungen mehreren Personen zugeordnet sind.

```
SELECT
    ex.id buchung_id, ...
    p.id person_id, ...
    bv.id bv_id, ...
    f.id fin_id, ...
FROM externebuchungsap ex
    JOIN externebuchungfinanzsapzuordnung z
        ON ex.id = z.buchung_id
    JOIN finanzierungsap f
        ON f.id = z.finanzierung_id
    JOIN beschaeftigungsverhaeltnis bv
        ON f.bvid = bv.id
    JOIN person p
        ON bv.personid = p.id
WHERE ex.id IN (
        SELECT z1.buchung_id
        FROM finanzierungsap f1
            JOIN externebuchungfinanzsapzuordnung z1
                ON f1.id = z1.finanzierung_id
            JOIN beschaeftigungsverhaeltnis bv1
                ON f1.bvid = bv1.id
            JOIN person p1
                ON p1.id = bv1.personid
        WHERE EXISTS (
                SELECT 1
                FROM finanzierungsap f2
                    JOIN externebuchungfinanzsapzuordnung z2
                        ON f2.id = z2.finanzierung_id
                    JOIN beschaeftigungsverhaeltnis bv2
                        ON f2.bvid = bv2.id
                    JOIN person p2
                        ON p2.id = bv2.personid
                WHERE z2.buchung_id = z1.buchung_id
                    AND bv1.id != bv2.id
                    AND f1.id != f2.id
                    AND p1.id != p2.id
            )
    )
ORDER BY ex.id, p.id, bv.id, f.id
```

Formulierungsvarianten des Prompts

- **Deskriptiv:** Ermittle all die Buchungs-IDs mit zugehörigen Personen-IDs, Beschäftigungsverhältnis-IDs und Finanzierungs-IDs, die über unterschiedliche Finanzierungen mehreren Beschäftigungsverhältnissen, Finanzierungen und Personen zugeordnet sind.
- **Technisch:** Selektiere aus ExterneBuchung, Finanzierung, Beschaeftigungsverhaeltnis und Person, wo mehrere Finanzierungen mit unterschiedlichen Personen derselben Buchung zugeordnet sind.
- **Zielbezogen:** Welche Buchungen, inklusive zugehöriger Person, Beschäftigungsverhältnis und Finanzierung, sind mehreren Personen über unterschiedliche Finanzierungen zugeordnet?
- **Kurzform:** Buchung, Person, Beschäftigungsverhältnis, Finanzierung, mehrfachen unterschiedlichen Personen über Finanzierung zugeordnet

B Verwendete R-Skripte

B.1 Boxplot der LLMs

```
library(ggplot2)
library(dplyr)

df <- read.csv("output.csv")

df %>%
  group_by(llm_name, sample_query_name) %>%
  summarise(avg_score = mean(score), .groups = "drop") %>%
  ggplot(aes(x = llm_name, y = avg_score)) +
  geom_boxplot() +
  stat_summary(fun = mean, geom = "point", shape = 20, size = 2, color = "red") +
  labs(x = "", y = "Score") +
  coord_cartesian(ylim = c(0.4, 1)) +
  theme(
    axis.text.x = element_text(size = 16),
    axis.text.y = element_text(size = 16),
    axis.title.x = element_text(size = 0),
    axis.title.y = element_text(size = 0)
  )
```

B.2 Boxplot der Komplexitäten

```
library(ggplot2)
library(dplyr)

df <- read.csv("output.csv")
df$sample_query_complexity <- factor(df$sample_query_complexity, levels
    = c("Einfach", "Mittel", "Komplex"))

df %>%
  group_by(sample_query_complexity, sample_query_name) %>%
  summarise(avg_score = mean(score), .groups = "drop") %>%
  ggplot(aes(x = sample_query_complexity, y = avg_score)) +
  geom_boxplot() +
  stat_summary(fun = mean, geom = "point", shape = 20, size = 2, color = "red") +
  labs(x = "", y = "Score") +
  coord_cartesian(ylim = c(0.55, 1)) +
  theme(
    axis.text.x = element_text(size = 16),
    axis.text.y = element_text(size = 16),
    axis.title.x = element_text(size = 0),
    axis.title.y = element_text(size = 0)
  )
```

B.3 Boxplot der Promptformulierungsarten

```
library(ggplot2)
library(dplyr)

df <- read.csv("output.csv")
df$prompt_type_name <- factor(df$prompt_type_name, levels
    = c("Deskriptiv", "Technisch", "Zielbezogen", "Kurzform"))

df %>%
  group_by(prompt_type_name, sample_query_name) %>%
  summarise(avg_score = mean(score), .groups = "drop") %>%
  ggplot(aes(x = prompt_type_name, y = avg_score)) +
  geom_boxplot() +
  stat_summary(fun = mean, geom = "point", shape = 20, size = 2, color = "red") +
  labs(x = "", y = "Score") +
  coord_cartesian(ylim = c(0.5, 1)) +
  theme(
    axis.text.x = element_text(size = 14),
    axis.text.y = element_text(size = 16),
    axis.title.x = element_text(size = 0),
    axis.title.y = element_text(size = 0)
  )
```

B.4 Heatmap nach LLM und Komplexität

```
library(ggplot2)
library(dplyr)

df <- read.csv("output.csv")
df$sample_query_complexity <- factor(df$sample_query_complexity, levels
    = c("Komplex", "Mittel", "Einfach"))

df %>%
  group_by(sample_query_complexity, llm_name) %>%
  summarise(score = mean(score), .groups = "drop") %>%
  ggplot(aes(x = llm_name, y = sample_query_complexity, fill = score)) +
  geom_tile(color = "white") +
  geom_text(aes(label = round(score, 2)), color = "black", size = 6) +
  scale_fill_gradient(low = "lightblue", high = "darkred") +
  labs(x = "", y = "") +
  theme(
    axis.text.x = element_text(angle = 45, hjust = 1, size = 14),
    axis.text.y = element_text(size = 16),
    axis.title.x = element_text(size = 0),
    axis.title.y = element_text(size = 0)
  )
```

B.5 Heatmap nach Komplexität und Promptformulierungsart

```
library(ggplot2)
library(dplyr)

df <- read.csv("output.csv")
df$sample_query_complexity <- factor(df$sample_query_complexity, levels
    = c("Komplex", "Mittel", "Einfach"))
df$prompt_type_name <- factor(df$prompt_type_name, levels
    = c("Deskriptiv", "Technisch", "Zielbezogen", "Kurzform"))

df %>%
  group_by(sample_query_complexity, prompt_type_name) %>%
  summarise(score = mean(score), .groups = "drop") %>%
  ggplot(aes(x = prompt_type_name, y = sample_query_complexity, fill = score)) +
  geom_tile(color = "white") +
  geom_text(aes(label = round(score, 2)), color = "black", size = 6) +
  scale_fill_gradient(low = "lightblue", high = "darkred") +
  labs(x = "", y = "") +
  theme(
    axis.text.x = element_text(angle = 45, hjust = 1, size = 14),
    axis.text.y = element_text(size = 16),
    axis.title.x = element_text(size = 0),
    axis.title.y = element_text(size = 0)
  )
```

B.6 Heatmap nach LLM und Promptformulierungsart

```
library(ggplot2)
library(dplyr)
library(reshape2)

df <- read.csv("output.csv")

df %>%
  group_by(llm_name, prompt_type_name) %>%
  summarise(score = mean(score), .groups = "drop") %>%
  ggplot(aes(x = prompt_type_name, y = llm_name, fill = score)) +
  geom_tile(color = "white") +
  geom_text(aes(label = round(score, 2)), color = "black", size = 6) +
  scale_fill_gradient(low = "lightblue", high = "darkred") +
  labs(x = "", y = "") +
  theme(
    axis.text.x = element_text(angle = 45, hjust = 1, size = 14),
    axis.text.y = element_text(size = 16),
    axis.title.x = element_text(size = 0),
    axis.title.y = element_text(size = 0)
  )
```

B.7 Facettiertes Säulendiagramm

```
library(ggplot2)
library(dplyr)

df <- read.csv("output.csv")
df$sample_query_complexity <- factor(df$sample_query_complexity, levels
    = c("Einfach", "Mittel", "Komplex"))

df %>%
  group_by(llm_name, prompt_type_name, sample_query_complexity) %>%
  summarise(score = mean(score), .groups = "drop") %>%
  ggplot(aes(x = prompt_type_name, y = score, fill = score)) +
  geom_col() +
  geom_text(aes(label = round(score, 2)), vjust = -0.5, size = 3) +
  facet_grid(sample_query_complexity ~ llm_name) +
  scale_fill_gradient(low = "lightblue", high = "darkred") +
  scale_y_continuous(expand = expansion(mult = c(0, 0.1))) +
  labs(x = "Prompt-Typ", y = "Score", fill = "Score") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1),
        strip.text = element_text(size = 10))
```