Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Entwicklung eines KI-Systems zur Schnittstellenanbindung von Auftragseingaben aus Fremdsystemen in FLEX

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.
Im Rahmen der Erstellung dieser Arbeit wurde das KI-System "GitHub Copilot" unterstützend zur sprachlichen Überarbeitung sowie zur fachlichen Reflexion und Präzisierung eigenständig entwickelter Argumente genutzt. Dieses System ist datenschutzkonform und kann sicher für vertrauliche Textpassagen im Rahmen interner Zwecke der Firma INFORM verwendet werden. Eine Übernahme von KI-generierten Texten oder inhaltlichen Lösungsvorschlägen erfolgte nicht. Sämtliche fachlichen Aussagen, Bewertungen und Schlussfolgerungen wurden eigenständig erarbeitet und verantwortet. Die Nutzung erfolgte im Einklang mit der Zweckbestimmung des Systems sowie unter Beachtung datenschutz- und urheberrechtlicher Vorgaben.
Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Joshua Paul Humberg
Aachen, den 15.12.2025

Unterschrift der Studentin / des Studenten

Entwicklung eines KI-Systems zur Schnittstellenanbindung von Auftragseingaben aus Fremdsystemen in FLEX

im Studiengang Angewandte Mathematik und Informatik

Name:	Joshua Paul Humberg
Matrikelnummer:	3651937
Professsor:	Prof. Dr. Alexander Voß
Betreuer:	Kai Ebenhöh M.Sc.
Unternehmen:	INFORM GmbH

5. Semester: Seminararbeit (Aachen) 15. Dezember 2025

Inhaltsverzeichnis

1 Abstract **1**

2 Einleitung **1**

3 Anforderungsanalyse **2**
3.1 Funktionale Anforderungen 2
3.2 Nicht-funktionale Anforderungen 3

4 Methoden und Verfahren **4**
4.1 Large Language Model 4
4.2 Prompt-Engineering 5
4.3 RAG 5
4.4 Instruction Prompt 6

5 Entwicklung **7**
5.1 Ansatz 7
5.1.1 Instruction-Prompt Erkenntnisse 9
5.2 Reflexion und Fokussetzung 10
5.3 Umsetzung in Visual Studio Code 11
5.3.1 Finaler Prompt 11
5.3.2 Kontextanreicherung 13
5.3.3 Interaktive Ausführung 14
5.4 Wahl des LLMs 15

6 Qualitätssicherung **16**
6.1 Bewertungsverfahren 16
6.2 Menschliches Feedback 16
6.3 Automatische Bewertung 17
6.4 Unittests 18

7 Ausblick **19**
7.1 Fine-Tuning 19
7.1.1 Abwägung Supervised Finetuning 20
7.1.2 Abwägung Reinforcement Learning from Human Feedback . . . 20
7.1.3 Finetuning Kosten 21
7.2 Skalierbarkeit 22
7.3 Integration 23

8 Diskussion **23**

9 Fazit **24**

A Anhang: Konkretes Beispiel-Mapping **25**

1 Abstract

Diese Arbeit evaluiert die Idee, eine KI-gestützte Teilautomatisierung der Schnittstellenentwicklung durch gezielte Kombination von Schema-Mapping[1] und Codegenerierung zu ermöglichen. Der Fokus liegt auf der Generierung eines zuverlässigen Mappings aus verschiedenen Eingangsformaten auf eine feste Zielentität sowie den dafür erforderlichen Anforderungen. Ein darauf spezialisierter Agent soll Informationen aus relevanten Feldern extrahieren und anschließend mithilfe probabilistischer Modellvorhersagen den passenden Zielfeldern zuweisen. Durch manuelle Eingriffe soll das erzeugte Mapping weiter präzisiert werden können. Als Bewertungskriterien werden die Einhaltung der Feldzuordnung und sie semantische Korrektheit des Outputs herangezogen.

2 Einleitung

Die INFORM Institut für Operations Research und Management GmbH entwickelt Softwarelösungen für die Optimierung von Geschäftsprozessen auf Basis von Operations Research und Künstlicher Intelligenz. Ihre verschiedenen Geschäftsbereiche decken Branchen wie Luftfahrt, Logistik, Finanzwesen und Telekommunikation ab, mit dem Ziel, Planungs- und Entscheidungsprozesse zu unterstützen und zu verbessern. Im Geschäftsbereich Industrielogistik und Healthcare der INFORM GmbH werden Softwareprodukte entwickelt, die innerlogistische Transport- und Logistikprozesse optimieren. Dabei gehört die Überwachung von Transportaufträgen zu den zentralen Prozessen und wird bei sämtlichen Kunden genutzt. Diese Transportaufträge können direkt im System angelegt werden, oder sie werden aus Fremdsystemen wie SAP importiert. Für die Auftragsimporte kommen aktuell manuell implementierte Schnittstellen zum Einsatz, um spezifische Anwendungsfälle für jeden Kunden abzubilden. Die empfangenen Nachrichten unterscheiden sich bei jedem Kunden zusätzlich in ihrer Datenstruktur und Mapping-Logik. Insgesamt existiert eine Vielzahl an Schnittstellen, deren Entwicklung und Pflege einen entsprechend hohen Ressourcenaufwand erfordert. FLEX ist in diesem Kontext eine Neuentwicklung des bestehenden Produkts, die sowohl in den Prozessen als auch in den Schnittstellen weniger projektspezifische Entwicklungen benötigt und einen standardisierten Weg der Produktanpassung bereitstellen soll. Für die Auftragseingabe über Schnittstellen sollen JavaScript basierte Mappings eingesetzt werden. Dadurch soll die Arbeitsbelastung beim Entwicklungsprozess verringert werden. Zur Realisierung dieses Bedürfnisses bieten sich KI-Lösungen an, da sie Routineaufgaben übernehmen können [von Richthofen et al., 2022].

[1] Unter Schema-Mapping versteht man die systematische Zuordnung von Feldern und Strukturen zwischen unterschiedlichen Datenmodellen auf ein Zielschema[Fagin, 2009]

3 Anforderungsanalyse

Um eine zielgerichtete Strategie zu entwickeln, müssen zunächst die Anforderungen an den Agenten genau definiert werden.

3.1 Funktionale Anforderungen

Die primäre Anforderung besteht darin, ein korrektes Mapping für einen Dokumenttyp zu gewährleisten, das anschließend auf mehrere Dokumente desselben Typs übertragbar ist. Auf dieser Grundlage ergeben sich die folgenden Teilaufgaben.

- **Verständnis der abzubildenden Entität**
  Das System muss das komplette Schema der Zielentität kennen. Das Schema enthält Informationen über sämtliche Pflichtfelder und optionale Felder der Zielentität. Dieses Wissen muss jederzeit abrufbar sein, damit alle folgenden Schritte sauber funktionieren.

- **Verarbeitung unterschiedlicher Input-Datenformate**
  Das System muss verschiedene eingehende Formate (XML, JSON und Text) verarbeiten und diese anhand eines definierten Schlüsselfeldes jeweils festen Mappingregeln zuordnen. Das Schlüsselfeld ist bei IDOC-Dateien (Eingangsformat aus SAP) am Tag „ICOCTYP“ erkennbar.

- **Automatische Generierung von Vorschlags-Mappings und JavaScript-Code**
  Das System muss für relevante Felder im Input-Datensatz automatisch eine passende Zuordnung zum Feld des Entität-Schemas vorschlagen, basierend auf Feldnamen, Typen, Semantik und Beispieldaten. Daraus entstehen initial für jeden Mappingtyp feste Mappingregeln. Diese Regeln sollen auf ein JavaScript Template angewendet werden, um ein ausführbares Snippet zu Testzwecken zu erhalten.

- **Verarbeitung expliziter Mappingregeln**
  Das System muss explizite Mappingregeln einlesen und auf die Zielentität anwenden können. Der resultierende Output eines spezifischen Inputs muss in einem passenden Format dargestellt werden. Dadurch sind die Feldbelegungen nachvollziehbar und es kann genau getestet werden.

- **Möglichkeit zum manuellen Eingriff**
  Bei Abweichungen sollen manuelle Texteingaben von Experten die Mappingregeln korrigieren. Wenn möglich, soll das System aus den manuellen Mappingvorschlägen lernen und bei zukünftigen Berechnungen eine höhere Präzision erreichen.

3.2 Nicht-funktionale Anforderungen

Obwohl der Agent zunächst zur Analyse und Korrektur von Mappingstrukturen entwickelt wird und noch nicht in ein produktives System integriert ist, müssen bestimmte Qualitätskriterien bereits eingehalten werden.

- **Bewertung des Mappings**
  Der Fokus der Arbeit liegt auf der Qualität des Mappings. Daher wird eine Möglichkeit gesucht, das Mapping automatisiert zu bewerten. Zu den Bewertungskriterien zählen Konsistenz, Reproduzierbarkeit sowie die Genauigkeit der Zuordnungen. Bei einem Agenten, der mit LLMs arbeitet, enstehen nicht immer reproduzierbare Ergebnisse, da ein LLM zwischen vielen möglichen Entscheidungspfaden wählen kann. So kann es zu dem Phänomen von verschiedenen Ergebnissen bei gleichem Prompting kommen, was bei Datenmapping unerwünscht ist. Auf diesen wichtigen Punkt wird im Kapitel Qualitätssicherung genauer eingegangen. Die Modellkorrektheit bzw. Zuverlässigkeit muss ebenfalls sichergestellt werden auf Grundlage der Modellgetriebenen Entwicklung (MDE). Für das Testen von exemplarischen Ergebnissen werden vorhandene Integrationstests verwendet, auf weitere Bewertungsmethoden wird auch eingegangen.

- **Lernfähigkeit**
  Der Agent soll Rückmeldungen verarbeiten und seine Entscheidungsregeln adaptiv weiterentwickeln.

- **Sicherheit**
  Die verarbeiteten Daten sollten bestmöglich keine sensiblen Informationen enthalten oder es sollte eine Umgebung genutzt werden, die ausreichend Datenschutz bietet (z.B. Copilot Enterprise).

- **Laufzeit und Kosten**
  Die Berechnungsdauer kann zunächst vernachlässigt werden, da der Agent in der Entwicklungsphase nicht zur Laufzeit in einem System eingebunden ist. Die Kosten des verwendeten Modells müssen zwar berücksichtigt werden, fallen jedoch bei ersten Testläufen mit GPT4.1 gering aus. Der Preis betrug 4 Euro für 2 Millionen verbrauchte Tokens (Begriff im Folgenden erläutert) in zahlreichen Testläufen.

4 Methoden und Verfahren

Zur Umsetzung der gewünschten Anforderungen ist zu klären, welche Entwicklungsschritte und Methoden sinnvoll sind, um ein KI-basiertes Mappingsystem zu realisieren. Das Ziel ist es, ein in sich geschlossenes System zu entwerfen, das selbstständig arbeitet und sämtliche Anforderungen des definierten Use Cases im Datenmapping vollumfänglich abdeckt. Im Folgenden werden bewährte Vorgehensmodelle und bestehende Entwicklungsprojekte betrachtet, um eine Umsetzungsstrategie zu entwickeln.

4.1 Large Language Model

Für KI-gestütztes Datenmapping braucht man zunächst ein Large Language Model (LLM), weil es semantische Zusammenhänge und Ähnlichkeiten erkennen kann [Parciak et al., 2024]. LLMs basieren meist auf neuronalen Netzen [Vaswani et al., 2017] und sind in der Lage, natürliche Sprache zu verarbeiten und Wahrscheinlichkeitsverteilungen für das nächste Token vorherzusagen [Müller and Schneider, 2025]. Ein Token ist dabei die Einheit, in die das LLM sämtlichen Text aufteilt, um ihn zu verarbeiten [GeeksforGeeks, 2023]. Bei der Codegenerierung gehören dazu auch Symbole wie „;“ oder „{}“. Da LLMs probabilistische Entscheidungen treffen, können sie auch Syntaxvorgaben einhalten und sind für die Codegenerierung geeignet [Chen et al., 2021]. Dabei hängt die Wahl des optimalen LLMs stark vom Usecase ab, weil alle Modelle spezifische Stärken und Schwächen haben. Die LLMs unterscheiden sich grundlegend in ihrer Modellgröße (Wissensbasis), Kontextlänge (Maximale Speicherkapazität im künstlichen Gedächtnis) und Tokenanzahl (Komplexität der Berechnung des LLMs) der Prompts. Insbesondere die Kontextlänge ist oft ein limitierender Faktor bei Aufgaben wie Schema-Mapping, da bei der Verarbeitung langer Dokumente oftmals mehr relevante Informationen benötigt werden, als das LLM zu einem Zeitpunkt speichern kann [Shapkin et al., 2023].

[Buss and Safari, 2025] untersuchten bereits das KI-gestützte automatische Schema-Mapping. Sie stellten fest, dass ohne menschlichen Einfluss kein funktional einwandfreies Ergebnis automatisch generiert werden kann.
LLM-basierte Agentensysteme profitieren daher stark von hybriden Ansätzen, bei denen menschliches Feedback und Anweisungen das LLM ergänzen. Durch Einbeziehung von Anweisungen aus der natürlichen Sprache, etwa durch Prompt-Engineering, kann das LLM seine Aufgaben genauer verstehen [Neubauer, 2025].

4.2 Prompt-Engineering

Die Verwendung von Prompt Engineering ermöglicht eine hybride Entwicklung eines Agenten, bei der die Stärken des LLM mit gezielter menschlicher Steuerung kombiniert werden.
„Prompt Engineering bezeichnet die zielgerichtete Gestaltung von Eingabeprompts für Foundation Models (Oberbegriff für LLMs), um deren generative Fähigkeiten für spezifische Aufgaben und Anwendungskontexte nutzbar zu machen, ohne dabei die Modellparameter direkt zu verändern“ [Hödl, 2025][S. 29]. Dabei werden Formulierungen, Struktur, Kontext und zusätzliche Parameter genutzt, um das Modell gezielt zu steuern. Prompt Engineering spielt eine zentrale Rolle bei der praktischen Anwendung von Large Language Models, da kleine Änderungen im Prompt oft zu deutlich unterschiedlichen Ausgaben führen können.
Durch gezielte Anwendung von Chain-of-Thought Prompting kann man das LLM dazu bringen, mehr nach logischen Schlüssen wie ein Mensch zu denken
[Tong and Zhang, 2024]. Ein Beispiel für diese Methode - auch logical reasoning genannt - ist „Löse Schritt für Schritt“. Dies ist ein einfacher Weg zu genaueren Antworten. Ein mögliches Beispiel im Kontext wäre: „setze zunächst die eindeutigen Felder und versuche daraus die unsicheren Felder abzuleiten“.

Eine andere Möglichkeit, um dem Sprachmodell Muster beizubringen, ist das few-shot learning. Dabei promptet man beispielhafte Input-Output Paare, die dem Modell als Lernmaterial dienen [Tong and Zhang, 2024]. Leider passiert hier nur In-Context-Learning, es werden also keine Modellgewichte angepasst, und nach der Session geht das Wissen wieder verloren. Es gibt eine weitere Möglichkeit, Trainingsbeispiele als Lernhilfe zu nutzen, ohne den Kontext zu fluten.

4.3 RAG

Mit Retrieval-Augmented Generation (RAG) können externe Wissensquellen in den Generierungsprozess eines LLM eingebunden werden. Die Technologie ermöglicht es, beispielsweise Mappingregeln, Schemata oder Beispieldaten in einer Datenbank abzulegen und bei Bedarf kontextsensitiv nachzuladen. Die Speicherung erfolgt oft in vektorbasierten Datenbanksystemen, die für eine effiziente semantische Suche ausgelegt und auf große Datenmengen skalierbar sind [Ma et al., 2025]. Durch Berechnung der semantischen Ähnlichkeit von Anfrage und Speicher werden möglichst passende Teile aus der Datenbank extrahiert und in den Kontext geladen [Gao et al., 2023].
Der wesentliche Vorteil von RAG besteht darin, dass der Kontext dynamisch erweiterbar ist: Neue oder geänderte Mappingregeln, zusätzliche Beispieldateien oder aktualisierte Schemata können zur Laufzeit in die Wissensbasis aufgenommen werden [Ma et al., 2025].

So kann der Agent flexibel auf Änderungen reagieren, oder sie sogar selber vornehmen. Damit eignet sich RAG für Szenarien, in denen sich die zugrunde liegenden Daten häufig ändern und ein statisch trainiertes Modell schnell veralten würde.

In aktuellen Arbeiten wird RAG daher als zentrales Architekturprinzip betrachtet, da es die Stärken großer generativer Modelle mit präziser, abrufbarer Wissensrepräsentation verbindet und so eine skalierbare, wartbare Wissensintegration ermöglicht
[Gao et al., 2023].

4.4 Instruction Prompt

Der Instruction Prompt ist die grundlegende Handlungsanweisung für den Agenten und definiert sein Verhalten. Man kann hier direkt einen Teil vom Kontextspeicher für jeden später getätigten Prompt belegen. Wie Li et al. zeigen, können Instruction Prompts die Einhaltung von Anweisungen stabilisieren, wobei ein vollständig reproduzierbares Verhalten über mehrere Interaktionen hinaus nur eingeschränkt belegbar ist [Li, 2024].

Für die Anwendung ist es entscheidend, dass der Instruction Prompt präzise formuliert ist und dem Agenten eine Aufgabe eindeutig beschreibt. In der Regel ist ein einzelner Agent (Single-Agent System, SAS) mit klar abgegrenzter Zuständigkeit für eine spezifische Aufgabe am besten geeignet, da sich sein Verhalten so konsistent bleibt und sich leichter evaluieren lässt [Gao et al., 2025]. Sollen hingegen mehrere, deutlich unterschiedliche Aufgaben oder Teilprobleme abgedeckt werden (z.B. Vorverarbeitung, Mapping, Validierung), bietet sich der Einsatz eines Multi-Agenten-Systems (MAS) mit spezialisierten Teilagenten an, die jeweils eigene Instruction Prompts erhalten und über Schnittstellen miteinander interagieren. Der MAS Ansatz erzeugt erheblich mehr Komplexität und erreicht dafür höhere Genauigkeit und Robustheit bei komplexen Anforderungen [Gao et al., 2025].

Die beschriebenen Konzepte – große Sprachmodelle (LLMs)
[Bommasani et al., 2021], Prompt Engineering [Liu et al., 2023], spezialisierte Instruction Prompts sowie Retrieval-Augmented Generation (RAG)[Lewis et al., 2020] – gelten in Kombination als etablierter Standard bei der Entwicklung moderner KI-Systeme. Sie dienen dazu, den Aufgabenbereich der Modelle präzise einzugrenzen, die benötigten Wissensquellen strukturiert bereitzustellen und das Antwortverhalten in Richtung nachvollziehbarer, schrittweiser Schlussfolgerungen zu steuern. Die praktische Umsetzung des Prototyps baut konsequent auf diesen Methoden auf und überträgt sie auf den domänenspezifischen Anwendungsfall des Mapping-Agents.

5 Entwicklung

In diesem Kapitel wird die konkrete Entwicklung des KI-gestützten Mappingsystems dokumentiert. Es wird dargestellt, wie die vorgestellten Methoden und Verfahren in der Praxis angewendet und angepasst wurden.

5.1 Ansatz

Zu Beginn der Arbeit wurde eine firmeninterne Entwicklungsumgebung zur Konfiguration persönlicher KI-Chatbots genutzt.
Diese stellt Funktionen wie die Wahl des LLMs, des Instruction Prompts und die Einstellung eines Kreativitätsparameters (Temperatur) sowie zusätzliche Individualisierungsmöglichkeiten zur Verfügung. Die Temperatur steuert dabei den Grad der Zufälligkeit in den Modellantworten. Niedrige Werte führen zu eher deterministischen, reproduzierbaren Ausgaben, während höhere Werte kreativere, aber auch weniger vorhersagbare Antworten begünstigen [Holtzman et al., 2020].
Ein Vorteil dieser Umgebung liegt in der Datensicherheit, da sie in der Unternehmensinfrastruktur betrieben wird. Somit konnten bedenkenlos Experimente durchgeführt werden.
Die ersten Experimente nutzten lediglich Kontextmanipulation über den Instruction Prompt. Dort wurde der Input auf XML-Dateien und der Output auf Javascript Snippets ohne genaue Vorgaben festgelegt. Dem Agenten wurden zudem die Felder der Zielentität im Instruction Prompt mitgeteilt (Tabelle 1). .

Tabelle 1: Visualisierter Ausschnitt des Schemas `CreateCustomerOrderRequest`

Feld	Typ	required	Beschreibung
startPosition	string	ja	Code of the start position
targetPosition	string	ja	Code of the target position
amount	number	nein	Amount to transport
scanId	string	ja	Optional scan ID
remark	string	nein	Optional remark for the order
customValues	object	nein	Generic custom values as key-value pairs

Mit diesen minimalen Anweisungen konnte der Agent eine im Prompt mitgegebene XML Datei verarbeiten und ein erstes Snippet erzeugen.
Obwohl die generierten Outputs weder syntaktisch noch semantisch vollständig korrekt waren, war der Bezug zur gegebenen Anweisung erkennbar und zeigte das grundsätzliche Potenzial des Ansatzes (Abbildung 1).

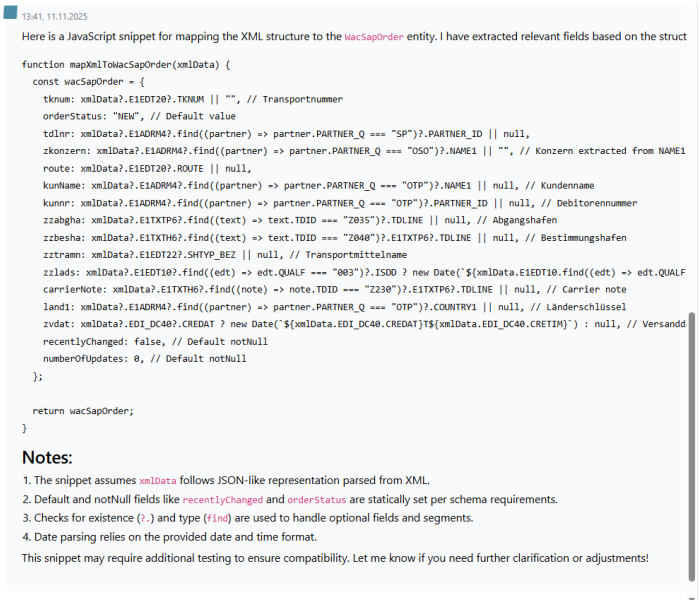

Abbildung 1: Diese erste Antwort verdeutlicht das grundsätzliche Potenzial des Ansatzes. Die JavaScript-Syntax wurde korrekt eingehalten und das erzeugte Ausgabeformat weicht nur geringfügig vom gewünschten Zielschema ab, obwohl noch kein Template vorgegeben war. Zudem wurden Begründungen für die Feldzuordnungen geliefert, auch wenn diese inhaltlich teilweise noch nicht korrekt waren

Es wurde jedoch schnell eine Limitierung der Inputlänge durch die vom LLM abhängige Kontextgröße identifiziert. Aufgrund der Größe der eingegebenen XML-Dateien wurde das Kontextfenster unabhängig vom ausgewählten LLM überschritten, was zu einer langen Rechenzeit ohne Ergebnis führte. Die meisten Informationen des Inputs sind zwar irrelevant, aber alle relevanten Felder müssen trotzdem erkannt und ausgelesen werden können. Daher setzt die Umsetzung der Aufgabe eine intelligente Kontextverwaltung voraus. Dafür wurde eine externe Datenbank als RAG-System angebunden. Die Dokumenteninhalte können bei dieser Technologie mittels Suche anhand der semantischen Ähnlichkeit abgerufen werden. Die RAG-Anbindung war für die ersten Experimente mit wenigen Dokumenten vollkommen ausreichend, es konnte jede Query sowie ihre Rückgabewerte eingesehen werden. Mit der Skalierung auf mehrere Dokumente im RAG wurden jedoch Schwachstellen identifiziert. So war etwa keine zuverlässige Dokumentsuche anhand des Dateinamens möglich, sodass das im Prompt explizit genannte Dokumente nicht in allen Fällen tatsächlich vom System herangezogen wurde. Die verwendete RAG-Technologie war nur für einen semantischen Vergleich der Anfrage mit den Inhalten, aber nicht mit den Namen der Dokumente ausgelegt. Um die Anforderung des Mappings umzusetzen, muss jedoch eine bestimmte Datei als Eingabe genutzt werden können, da sich die Zuordnung immer an der konkreten Eingabestruktur orientiert. Somit war diese Umgebung nicht gänzlich für die Anforderung geeignet. Es konnten jedoch wichtige Erkenntnisse über die praktische Umsetzung gewonnen werden:

5.1.1 Instruction-Prompt Erkenntnisse

Schon nach wenigen Tests wurde festgestellt, dass kleine Anpassungen im Instruction-Prompt das Ergebnis erheblich beeinflussen können und seine Konfiguration daher ein entscheidender Schritt in der Lösungsfindung ist. Auf Basis verschiedener getesteter Instruction-Prompts wurden folgende Erkenntnisse für eine systematische Weiterentwicklung gezogen:

- **Input-/Output festlegen** Der Prompt muss die erwarteten Eingabeformate und das gewünschte Ausgabeformat klar definieren, um dem Agenten Orientierung zu bieten. Aus Experimenten sowie einer externen Quelle geht hervor, dass die Einhaltung der Syntax am zuverlässigsten durch ein vorgegebenes Template und Beispiele im Kontext erreicht wird [Reynolds and McDonell, 2021].

- **Handlungsablauf** Der Prompt muss den generellen Handlungsablauf des Agenten Schritt für Schritt nach dem Chain-of-Thought Prinzip beschreiben, um eine konsistente und methodische Vorgehensweise zu gewährleisten. Soll innerhalb des RAG-Mechanismus auf Dateien zugegriffen werden, müssen die dafür geltenden Regeln explizit beschrieben werden.
  In Experimenten ohne klar vorgegebenen Handlungsablauf traten vermehrt Halluzinationen und inkonsistente Ergebnisse auf oder der Ablageort benötigter Dateien wurde nicht gefunden. Die logischen Schritte zur Ergebnisfindung müssen daher explizit im Prompt vorgegeben werden.

- **Sonderfälle** Ausnahmefälle und Randbedingungen müssen antizipiert und mit expliziten Handlungsanweisungen versehen werden, um Robustheit zu sichern.
  Im konkreten Anwendungsfall ist der Umgang mit nicht gesetzten Pflichtfeldern und möglichen Zusatzfeldern ein zu behandelnder Ausnahmefall, dessen Vernachlässigung dazu führte, dass generierte Zielobjekte entweder formell ungültig waren oder relevante Zusatzinformationen verloren gingen.

- **Spezialisierung** Ein Agent sollte auf eine spezifische Fähigkeit oder Aufgabe spezialisiert sein, anstatt mehrere Use Cases zu vereinen. Dies erhöht die Transparenz und vereinfacht die Weiterentwicklung.
  In den ersten Tests wurde dieser Aspekt nicht berücksichtigt: Der Agent sollte gleichzeitig das Mapping entwerfen und JavaScript-Code generieren. Dadurch war die Nachvollziehbarkeit des Outputs eingeschränkt und mögliche Fehlerursachen wurden nur schwer erkannt. Als Konsequenz wurden feste Mapping-Regeln als Zwischenschritt eingeführt, sodass die Codegenerierung erst auf Basis eines zuvor validierten Mappings erfolgt.

- **Formulierung** Alle Anweisungen müssen präzise und eindeutig formuliert sein, um Interpretationsspielräume und damit verbundene Fehlerquellen zu minimieren [White et al., 2023].

- **Prägnanz** Ein übermäßig langer und detaillierter Prompt kann die Ergebnisqualität verschlechtern, da er zu Redundanzen und Ablenkungen führt [Chatterjee, 2025]. Die Formulierung sollte präzise und auf das Wesentliche reduziert sein, um diese Halluzinationen zu vermeiden und Kontextspeicher einzusparen.

5.2 Reflexion und Fokussetzung

In Anbetracht der gewonnenen Erkenntnisse durch die ersten Experimente und der erkannten Schwachstelle der RAG-Suche war eine Neuausrichtung erforderlich.

In den ersten Experimenten wurde der Agent als ganzheitlicher Schnittstellengenerator konzipiert: Er sollte eine Eingabedateien analysieren, daraus ein Mapping ableiten und dieses direkt in lauffähigen FLEX-spezifischen JavaScript-Code überführen, der anschließend in der AWS-Laufzeitumgebung getestet wurde. Diese Herangehensweise erwies sich jedoch als problematisch, weil Fehlerquellen aus Mapping, Codegenerierung und Zielumgebung miteinander vermischt wurden. Zusätzlich erschwerte die langsame Ausführung und die nur indirekte Einsicht über Serverlogs die Analyse der Ergebnisse. Dadurch waren Transparenz und Nachvollziehbarkeit der Einzelschritte stark eingeschränkt und die gezielte Weiterentwicklung des Mappings wurde behindert. Aus diesen Schwachstellen wurde die Konsequenz gezogen, Parsing, Mapping und Codegenerierung klar zu trennen und den Fokus der Arbeit zunächst auf ein fachlich valides, lokal testbares Mapping zu legen, bevor die eigentliche Laufzeitintegration in FLEX betrachtet wird.

Die Fokussierung auf die Laufzeitintegration stellte somit einen Schritt vor der eigentlichen Kernaufgabe dar – der Erstellung eines korrekten und nachvollziehbaren Mappings. Daher wurde der Fokus der Arbeit bewusst auf die Entwicklung eines zuverlässigen und validen Mappings eingegrenzt. Dieses Mapping wird weiterhin in ausführbarem Code abgebildet, allerdings zunächst in einer vereinfachten, lokal testbaren Syntax (JavaScript-Snippets in der HTML-Testumgebung), bevor eine spätere Übersetzung in das endgültige FLEX-Kommando erfolgen kann. Außerdem musste aufgrund der eingeschränkten RAG-Suche eine Umgebung mit speziell dafür ausgelegter RAG-Integration genutzt werden.

5.3 Umsetzung in Visual Studio Code

Nach den ersten Experimenten in der firmeninternen Umgebung wurden die weiteren Untersuchungen zum Mappingsystem in Visual Studio Code durchgeführt. Dort kam GitHub Copilot als LLM-basierter Assistent zum Einsatz, der im Wesentlichen die gleichen Konfigurationsmöglichkeiten wie die vorherige Umsetzung anbietet.

Zusätzlich konnte standardmäßig eine gezielte Dateisuche durchgeführt werden, wodurch die vorherige Problematik behoben wurde. Darüber hinaus gestaltete sich die Navigation und Entwicklung in der Entwicklungsumgebung als benutzerfreundlich, da Kontext und Chat in einer Oberfläche integriert sind. Grundsätzlich wurde der gleiche Ansatz von Instruction Prompt und interner RAG-Anbindung genutzt. Eine entscheidende zusätzliche Möglichkeit liegt aber in der eigenständigen Manipulation von Kontext-Dateien und die damit einhergehende Realisierbarkeit einer iterativen Verbesserung durch Feedback.

5.3.1 Finaler Prompt

Der Finale Prompt ergibt sich aus den zuvor gesammelten Anforderungen und Methoden. Zunächst wird in der Einleitung die **Spezialisierung** des Agenten klargestellt:

> Du bist ein spezialisierter Mapping-Agent. Deine Aufgabe ist es, Felder beliebiger XML- oder JSON-Eingaben auf ein vorgegebenes Entitätschema abzubilden. Für jedes Zielfeld wählst du anhand von Feldnamen, Struktur, Kontext und deinem vorhandenen Wissen das passendste Eingabefeld aus und begründest deine Zuordnung, wenn sie nicht eindeutig ist.

Nachdem ausführliche Experimente gezeigt haben, dass eine Best-Practice bei der Agentenkonfiguration die Definition eines festen **Ablaufplans** darstellt, wurde dies auch umgesetzt. Unter Berücksichtigung der genannten Erkenntisse bezüglich **Prägnanz** und **Formulierung** wurde die generelle Vorgehensweise im Instruction Prompt final wie folgt definiert:

> Analysiere die Struktur des eingehenden Objekts. Nutze vorhandene Mapping-Regeln, wo sie definiert sind, und leite für alle übrigen Felder auf Basis von Feldnamen, Struktur, Kontext und deinem Wissen sinnvolle Zuordnungen ab. Nutze Feedback, um dein Mapping-Wissen und deine Heuristiken schrittweise zu verbessern. Ziel ist es, alle Felder des Entitätschemas möglichst sinnvoll und vollständig zu belegen – auch bei unbekannten oder neuen Formaten.

Für das Eingangsformat XML wurden die Anweisungen sowie die erwarteten **Input-/ Output** Formate nochmal präzisiert, damit die Vorgehensweise eindeutig ist:

> Erweiterung (IDoc-spezifische Regeln):
> 1. Bestimme den IDOCTYP aus dem XML.
> 2. Lade die zugehörige Mapping-Regeldatei, lege für neue oder unbekannte IDOCTYPS neue Regeldateien an, die du nach Feedback weiter verfeinerst.
> 3. Wende diese Regeln beim Feld-Mapping auf JavaScript-Templates an

Es fehlte noch eine genauere Anleitung zum Erstellen neuer Mapping-Regeln, was den Kern der Aufgabe des Agenten darstellt. Diese Regeln sind zwar umfangreicher definiert, lassen sich aber auf diese Prinzipien zurückführen:
Die Orientierung am Zielschema (CreateCustomerOrderRequest) bildet die Grundlage für die heuristischen Zuordnungen, da das Schema neben den Feldnamen und Datentypen auch kurze Beschreibungen zur Bedeutung der Felder enthält (Tabelle 1, S.7). In Kombination mit den Feldnamen und Datentypen der jeweiligen Eingabe können die verfügbaren Möglichkeiten zur Belegung jedes Feldes schon eingegrenzt werden.
Zudem muss der Agent aus allen bestehenden Beispielen und Regeln zusätzliche Schlüsse ziehen und diese auf neue Formate anwenden.
Im Falle einer Unsicherheit soll dies auch klar kommuniziert werden. Statt willkürlich Felder zu belegen, sollen mehrere Möglichkeiten gegeben und auf diese Weise eine höhere Transparenz erreicht werden.
Menschliches Feedback über den Chat soll flexibel in das System einfließen. Auf diese Weise soll die Zuordnung schrittweise verbessert werden können.
Um **Sonderfälle** abzudecken, wurden unbekannte Zusatzfelder zunächst unverändert in das Zielobjekt übernommen, während Pflichtfelder bei fehlender Belegung mit klar erkennbaren Defaultwerten versehen wurden, um in jedem Fall ein formal valides Zielobjekt zu erzeugen. Dieses Verhalten ist anpassbar, für die durchgeführten Tests erwiesen sich Defaultwerte jedoch als zweckmäßig, um fehlende Pflichtfelder zu identifizieren.

Durch die Ablage des Prompts im Projektkontext konnte sein Inhalt durch Prompteingaben (im Kapitel Kontextanreicherung erläutert) iterativ auf Basis der gewonnenen Erkenntnisse und Testergebnisse angepasst werden. Konkret wurden nach fehlgeschlagenen oder unvollständigen Mappings zusätzliche Regeln und Präzisierungen ergänzt - die Verpflichtung zur Nutzung expliziter Mapping-Regeldateien, Anweisungen zum Umgang mit Pflicht- und Zusatzfeldern, ein fest definierter Handlungsablauf sowie Anhaltspunkte zur Verbesserung des Initialmappings. Diese schrittweisen Änderungen führten dazu, dass der Agent sukzessive weniger Halluzinationen zeigte und die Felder

des CreateCustomerOrderRequest nachvollziehbarer belegte. Auf diese Weise entstand eine Evolution der Konfiguration, durch die der Agent immer genauer auf seine spezifische Mappingaufgabe kalibriert wurde.

5.3.2 Kontextanreicherung

Die Kontextanbindung ist in VS Code ohne großen Initialaufwand über das integrierte RAG-System möglich. Alle relevanten Dateien liegen im selben Projektverzeichnis und sind damit für den Agenten zugänglich. Über die semantische Suche wurden mit hoher Trefferquote gegebene Dateien bei Bedarf in den Kontext geladen. Die Limitierung der Kontextgröße aufgrund langer Inputdateien konnte so gelöst werden. Die zuvor ausgeführte Evolution der Konfiguration resultiert aus der zusätzlichen Möglichkeit zur direkten Kontextmanipulation durch Nutzereingaben.
Ein kleines Hindernis stellte die Berechtigung des Agenten auf automatischen Zugriff auf den Projektordner aus Datenschutzgründen dar. Er musste oft mehrfach aufgefordert werden, die Dateien zu laden, was den Entwicklungsprozess verlangsamte.
Dies konnte aber durch folgendes Statement im Instruction Prompt größtenteils umgangen werden, sodass zu Beginn eines neuen Chats nur noch eine Bestätigung nötig ist.

> Du darfst auf alle Dateien im Ordner „Agent_Files“ zugreifen und sollst zu Beginn eines neuen Chats einmal kurz nachfragen, ob dieser Zugriff erlaubt ist.

Es gibt zusätzlich die Möglichkeit, einzelne Dateien explizit in den Kontext zu laden.

Folglich lag der Fokus weniger auf der technischen Einbindung und stärker auf der inhaltlichen Auswahl eines sinnvollen Kontexts.
Es ist unerlässlich, als zentralen Ausgangspunkt den Instruction Prompt einzubinden. Zu Beginn wurde eine Input-XML sowie das Kommando-Schema mitgegeben. Daraufhin wurden erste Mappingregeln im CSV Format generiert, mit Hilfe derer ein erster JSON-Output erstellt wurde. Auf diesen Grundlagen aufbauend wurde der Kontext durch weitere Befehle iterativ weiterentwickelt. Eine Speicherung von neuen Regeldateien und Outputs erfolgt dank dieser Anweisung immer automatisch.

> Neue oder geänderte Mapping-Regeln sollen automatisch als CSV im Ordner „mappingregeln“ gespeichert werden. Für jedes bearbeitete Eingabebeispiel ist automatisch ein entsprechender Output im Zielschema als JSON im Ordner „Outputs“ abzulegen (Dateiname analog zur Inputdatei).

Je mehr Beispiele vorliegen, desto mehr Orientierungsmöglichkeiten stehen dem Agen-

ten zur Verfügung. Letztlich wurde sich auf folgende erweiterbare Struktur festgelegt.

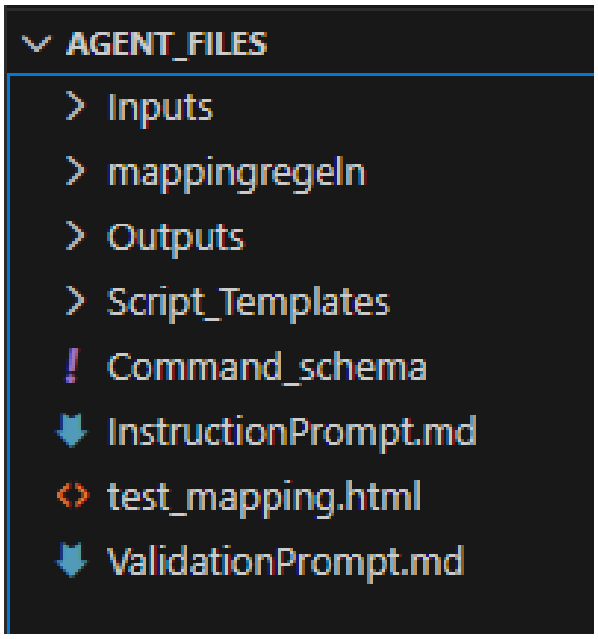

Abbildung 2: Ordnerstruktur im Projektkontext. Die Inputs enthalten Eingangsdokumente und in den Outputs werden die generierten Skripte gespeichert. Im Kapitel Qualitätssicherung wird näher auf den Hintergrund des Htmls und ValidationPrompts eingegangen.

5.3.3 Interaktive Ausführung

Um den Agenten zur Ausführung der Aufgabe zu veranlassen, muss stets ein Prompt in VS Code angestoßen werden. Es wurde sich auf zwei verschiedene Arten von Anweisungen festgelegt.

Zum einen soll der Agent eine eingehende Datei beliebigen Formates direkt in ein JSON-Objekt der Zielentität überführen können. Dies ist vor allem bei der Entwicklungsphase eine praktische Möglichkeit, um ohne Aufwand direkt im Chat ein Mapping zu testen.

Für die angestrebte Lösung der Schnittstellen soll jedoch nicht für jedes einzelne Eingangsdokument eine direkte Berechnung durch das LLM erfolgen. Dies wäre einerseits zu rechenintensiv und andererseits nicht zuverlässig einsetzbar, da ein LLM ein probabilistisches Modell bleibt und selbst bei einem optimierten Instruction Prompt noch mit Halluzinationen und potenziell fehlerhaften Ergebnissen zu rechnen ist.
Daher wurde der Standardmodus des Agenten so konfiguriert, dass er wiederverwendbare Javascript-Mappings erzeugt. Diese Mappings orientierten sich zunächst an einer

vereinfachten, testbaren Syntax. Zunächst wurde die in FLEX produktiv eingesetzte, lauffähige Syntax übernommen. Aufgrund der in AWS eingeschränkten Testbarkeit war jedoch eine leicht abgewandelte Syntax erforderlich, um die Tests auf der lokalen HTML-Seite effizienter und schneller durchführen zu können. Anhand der iterativ entwickelten deterministischen Mappings und eines Javascript-Templates wird für jeden Dokumenttypen statt für jede Datei ein Script erstellt. Die Scripte werden anschließend für gleichartige Dokumente wiederverwendet. Im Anhang A wird ein konkretes Mapping gezeigt.

5.4 Wahl des LLMs

Darüber hinaus wurde im Rahmen der Testläufe auch die Auswahl des zugrunde liegenden LLMs untersucht, da sich dabei neue Erkenntnisse ergaben.
Obwohl das in Visual Studio Code verwendete Sprachmodell grundsätzlich flexibel gewählt werden kann, zeigte sich in den Experimenten insbesondere GPT-4.1 als zuverlässig, da es die vorgegebenen Templates konsistent einhielt.
Bei der Nutzung der neueren Variante GPT-5.1 (Preview) traten hingegen vermehrt Verstöße gegen die Templatevorgaben auf. Das Modell halluzinierte vermeintlich notwendige Hilfsfunktionen, welche die Ausführbarkeit der erzeugten Skripte eher beeinträchtigten. Dieses Verhalten ist im Kontext der Arbeit unerwünscht, weshalb im Folgenden mit GPT-4.1 gearbeitet wurde. Eine ausführliche Auswahl und Bewertung des am besten geeigneten LLMs für das Schema-Mapping, insbesondere im Hinblick auf semantische Ähnlichkeiten, konnte im begrenzten Zeitrahmen und durch mangelnde Validierungsdaten nicht vorgenommen werden.

6 Qualitätssicherung

Die ausführlich durchgeführte Qualitätssicherung setzt sich aus mehreren Schichten zusammen, die kombiniert eine genaue Bewertung der Mappings sowie der Codegenerierung möglich machen. Dabei wurde besonders Wert auf die semantische Korrektheit gelegt. Die syntaktische Korrektheit war durch die Verwendung fester Script-Templates in der Regel sichergestellt und stellte eine vergleichsweise kleine Herausforderung dar. Zunächst wird auf verbreitete Bewertungsverfahren eingegangen und ihre Umsetzbarkeit diskutiert.

6.1 Bewertungsverfahren

Die **testbasierte Bewertung** zeichnet sich durch einen handgeschriebenen Testsatz für jedes Problem aus. Mit ausreichender Testabdeckung kann so eine Funktion sichergestellt werden [**?**], jedoch ist der Initialaufwand für die Unittests sehr hoch. Da sich FLEX noch nicht im produktiven Einsatz ist, existieren keine echten Beispiele.

Eine weitere, besonders im KI-Kontext verbreitete Methode ist die **tokenbasierte Bewertung**. Dabei wird der generierte Code mithilfe eines Tokenizers in Tokens zerlegt und mit einer bekannten Musterlösung verglichen. Bewertet wird dann die Anzahl der abweichenden Tokens im Output [Tong and Zhang, 2024]. Dieses Verfahren setzt jedoch valide Ergebnisse in tokenisierter Form voraus, sodass es im Rahmen dieser Arbeit nicht eingesetzt werden konnte.

Da sowohl testbasierte als auch tokenbasierte Bewertungsverfahren im vorliegenden Use Case nur eingeschränkt einsetzbar sind, wurde ein mehrstufiges Qualitätssicherungskonzept entwickelt, das menschliches Feedback, eine automatische Validierungsschicht und exemplarische Unittests kombiniert.

6.2 Menschliches Feedback

Die hybride Entwicklung kombiniert automatisierte Vorschläge des Mapping-Agents mit gezielter menschlicher Überprüfung. Fachliche Rückmeldungen werden genutzt, um Fehler und unerwünschtes Verhalten zu identifizieren und die Mapping-Regeln iterativ zu verfeinern. Konkret werden neue Initialmappings manuell geprüft. Auf Basis dieser Prüfung wird dem Agenten mitgeteilt, welches Zielfeld aus welcher Quelle belegt sein soll und welche Zuordnung fehlerhaft war. Diese Rückmeldung fließt anschließend in die Aktualisierung der Mapping-Regel für den spezifischen Dokumententyp ein. Durch diese wiederholte Schleife aus Agentenvorschlag, automatischer Validierung und menschlichem Feedback entsteht ein kontinuierlicher Verbesserungsprozess. Die hybride

Entwicklung erhöht damit sowohl die semantische Korrektheit der Mappings und bietet zusätzliche Sicherheit gegenüber einer rein automatisierten Lösung.

Um diese manuelle Überprüfung zu ergänzen und zu entlasten, wurde zusätzlich eine automatische Bewertungsschicht in Form eines Validierungs-Agenten eingeführt.

6.3 Automatische Bewertung

Zusätzlich wurde eine automatisierte Validierungsschicht über einen unabhängigen zweiten Agenten im „ValidationPrompt.md“ definiert, um die Ergebnisqualität der Codegenerierung abzusichern.

Eine vollständig automatisierte Bewertung des Mappings ist im betrachteten Use Case nicht möglich, da zuvor unbekannte Zusammenhänge ermittelt werden, für die kein Validierungsdatensatz existiert. Stattdessen unterstützt der Validierungs-Agent die hybride Entwicklung, indem er für jedes generierte Snippet prüft, ob die definierten Mapping-Regeln eingehalten werden und der Output das Zielschema erfüllt.
Um diese Genauigkeit zu erreichen, prüft der Validierungsagent für jedes generierte Snippet die Einhaltung folgender Kriterien:

- Er kontrolliert feldweise, ob das vom Mapping-Agenten erzeugte Ergebnis die vorliegenden Mapping-Regeln korrekt einhält.
- Zusätzlich stellt er sicher, dass der Output ein syntaktisch gültiges Objekt der Zielentität **CreateCustomerOrderRequest** darstellt und damit das zugrunde liegende Schema erfüllt. Wenn in angewendeten Mapping-Regeln Pflichtfelder nicht gesetzt werden, macht der Agent darauf aufmerksam und sichert so das Mapping ab.

In den durchgeführten Tests hat der Validierungs-Agent alle generierten Ergebnisse akzeptiert, was darauf schließen lässt, dass die Codegenerierung auf Basis definierter Vorlagen (JavaScript-Template und Mapping-Regeln) mit hoher Genauigkeit funktioniert.
Natürlich kann auch dieser Agent halluzinieren, er stellt jedoch einen wichtigen Schritt hin zu einem syntaktisch validen Output dar.

Neben dieser agentenbasierten Bewertung wurden darüber hinaus klassische Unittests eingesetzt, um das Verhalten der generierten Mappings zu überprüfen.

6.4 Unittests

Zur Bewertung der Qualität der generierten Objekte wurden acht bestehende Unittests aus Kundenumgebungen mit den generierten Ergebnissen verglichen und daraus die initiale Genauigkeit der Übereinstimmung mit den Testergebnissen für jedes gesetzte Feld berechnet.

Für die Ausführung der JavaScript-Snippets wurde eine separate Umgebung benötigt. Diese wurde als einfache HTML-Seite implementiert, welche die generierten JavaScript-Snippets ausführen kann. Die Snippets können als Eingabe ausschließlich JSON verarbeiten, da sich eine direkte Verarbeitung von XML in JavaScript ohne zusätzliche Bibliotheken im Browser als nicht praktikabel erwies. Daher musste das XML zunächst in einem separaten Schritt in ein JSON-Objekt überführt werden. Daraus resultiert eine klare Trennung von Parsing- und Mapping-Schicht, was die unabhängige Testbarkeit verbessert [Parnas, 1972]. Exemplarisch wurden Tests aus drei unterschiedlichen Kundenumgebungen bestmöglich aufbereitet, damit sie auf das neue Entitätsschema anwendbar sind. Die Kundenumgebungen nutzen stark angepasste Varianten der behandelten Entität. Viele der geprüften Felder sind daher Sonderfälle, die im ursprünglichen FLEX-Schema nicht enthalten sind. Dadurch schneiden diese Testfälle im Initialmapping tendenziell schwächer ab.

Im Durchschnitt wurden die 20 manuell betrachteten Felder initial mit einer Genauigkeit von 30% wie im Test erwartet belegt.
Diese niedrige Trefferquote resultiert jedoch größtenteils aus der eingeschränkten Testbarkeit. Es wurde festgestellt, dass fünf erwartete Feldwerte im Input gar nicht vorhanden waren und stattdessen durch Defaultwerte oder zusätzliche Logik belegt werden. Unter Berücksichtigung dieses Umstands erhöht sich die erzielte Genauigkeit bereits auf $6/15 = 40\%$.
Die verbleibenden Felder aus den Inputs wurden zunächst nicht korrekt erkannt, da sie unter einem anderen Feldnamen erwartet wurden oder eine Transformation wie das Aufteilen des Codes durch einen Bindestrich vorausgesetzt wurde. Die dafür notwendige Logik ist im Input nicht enthalten. Nach einer Iteration, in der diese Feinheiten per Prompt erläutert wurden und der Agent daraufhin die entsprechenden Regeln sowie Skripte angepasst hat, konnten diese Felder wie erwartet zugeordnet werden. Unter der idealisierten Annahme, dass menschliches Feedback fehlerfrei ist, ließe sich theoretisch nach genügend Iterationen eine Genauigkeit von 100% erreichen.
Die beschriebenen Tests decken nur eine begrenzte Anzahl von Beispielen ab und prüfen jeweils nur einen Teil der Felder der Zielentität. Daher sind sie im Sinne einer umfassenden Testabdeckung nicht vollständig aussagekräftig. Sie sind jedoch die einzige verfügbare Möglichkeit, erwartete Ergebnisse zu testen, und illustrieren exemplarisch

die Funktionsweise des Initialmappings und die Verbesserung durch iteratives Feedback.

Insgesamt entsteht durch die Kombination aus einer hybriden, iterativen Entwicklung der Mappingregeln sowie internem Validierungs-Agenten und zusätzlichen exemplarischen Integrationstests ein mehrstufiges Qualitätssicherungs-Konzept: auf der einen Ebene wird die fachliche Korrektheit des Mappings durch einen Menschen validiert; auf der anderen Ebene wird die Einhaltung dieses Mappings innerhalb der Agentenumgebung bei der Anwendung auf das JavaScript-Template überprüft.

7 Ausblick

Es existieren neben den angewendeten Verfahren weitere Möglichkeiten, ein LLM dauerhaft auf den speziellen Use Case des Schema-Mappings zu spezialisieren. Aufgrund des hohen zeitlichen Aufwands wurden diese Ansätze nicht umgesetzt, sondern lediglich konzeptionell betrachtet. Damit ist die Lösung mit vertretbarem Aufwand realisierbar, gleichzeitig bleibt ausreichend Spielraum für spätere Erweiterungen und Verbesserungen.

Eine mögliche Verbesserung stellt das so genannte Fine-Tuning dar, mit dem die Genauigkeit des Initialmappings weiter erhöht werden kann, weshalb es im Folgenden vorgestellt wird.

7.1 Fine-Tuning

Fine-Tuning ist ein Verfahren des Maschinellen Lernens, bei dem ein vortrainiertes LLM mit einem domänenspezifischen Trainingsdatensatz weitertrainiert wird, um seine Gewichte an einen speziellen Usecase anzupassen [Devlin et al., 2019]. Ein genau abgestimmtes LLM kann einen komplexen Prompt in seinem Fachgebiet deutlich schneller und mit niedrigerer Fehlerquote beantworten als ein lediglich mit Few-Shot-learning trainiertes Modell (Abbildung 4).

Method	MNLI-m (Val. Acc./%)	RTE (Val. Acc./%)
GPT-3 Few-Shot	40.6	69.0
GPT-3 Fine-Tuned	89.5	85.4

Abbildung 3: Fine-Tuning erzielt höhere Validierungsgenauigkeit als Few-Shot-Learning auf mehreren Datensätzen [Hu et al., 2021]

Es existieren verschiedene Fine-Tuning-Varianten, die unterschiedliche Ansprüche an Rechenleistung sowie passend zugeschnittene Trainingssätze haben [OpenAI, 2023] und

damit unterschiedlich hohe Kosten verursachen. Daher ist eine Analyse des Aufwand-Nutzen-Verhältnisses vor der Durchführung angebracht. Im Folgenden werden die in diesem Kontext naheliegendsten Varianten sowie ihr möglicher Mehrwert für das Schema-Mapping dargestellt.

7.1.1 Abwägung Supervised Finetuning

Ein gängiges Fine-Tuning-Verfahren ist das supervised Fine-Tuning (SFT), da es auf klar definierten Eingabe-Ausgabe-Paaren basiert und somit eine direkte Optimierung auf korrekte Schema-Mappings ermöglichen würde [OpenAI, 2023]. In einem stabilen Umfeld mit weitgehend unveränderten Formaten könnte ein so spezialisiertes Modell das Initialmapping die Fehlerrate deutlich reduzieren. Im betrachteten Use Case ändern sich die Eingabeformate jedoch regelmäßig, und für neue Varianten liegen zunächst keine verlässlichen Validierungsdaten in ausreichender Menge vor. Damit fehlt die zentrale Voraussetzung für SFT: ein umfangreicher, konsistenter Trainingsdatensatz mit korrekten Mappings. Ein derartiges Datenset müsste erst mit erheblichem manuellem Aufwand aufgebaut und kontinuierlich nachgepflegt werden, was den erwarteten Nutzen des Fine-Tunings für das Schema-Mapping wirtschaftlich relativiert. Das SFT würde hingegen eine genauere Einhaltung der Templates ermöglichen, da es vorgegebene Formate präzise einhalten kann [OpenAI, 2023]. Dieses Teilproblem der Codegenerierung wurde jedoch ohne Fine-Tuning schon ausreichend eingehalten.
Ein weiteres verbreitetes Verfahren ist das Reinforcement Learning.

7.1.2 Abwägung Reinforcement Learning from Human Feedback

Reinforcement Learning lässt sich als „a way of programming agents by reward and punishment without needing to specify how the task is to be achieved“ beschreiben [Kaelbling et al., 1996]. RLHF ersetzt dabei kein SFT, sondern baut auf einem bereits trainierten Modell auf und optimiert dessen Verhalten über Belohnungssignale. Daraus geht hervor, dass sich RLHF als nachgelagerter Optimierungsschritt für den Use Case anbietet, da es keine zusätzlichen expliziten Zielausgaben benötigt. Die Spezialisierung Reinforcement Learning from Human Feedback (RLHF) könnte im Kontext der hybriden Entwicklung genutzt werden, um Mapping-Entscheidungen auf Basis von Belohnungssignalen schrittweise zu verbessern und als Resultat für neue oder sich ändernde Typen genauere Initialmappings gewährleisten. Im vorliegenden Anwendungsfall wird ohnehin mit menschlichen Interaktionen gearbeitet, weshalb sich das RLHF grundsätzlich anbieten würde. Dabei wird das Fine-Tuning eher als langfristiger iterativer Prozess angesehen, bei dem zunehmend Feedback gesammelt wird und das Mapping dadurch genauere Ergebnisse erzielen kann. Die Anforderungen für eine mögliche Umsetzung werden im Folgenden diskutiert.

7.1.3 Finetuning Kosten

Die Umsetzung des RLHF baut auf der des Supervised Finetunings auf, es wird aber zusätzlich eine Präferenzschicht trainiert, in die das menschliche Feedback einfließt [Dai and Pan, 2023]. Eine genaue Betrachtung der Kosten würde den Rahmen der Arbeit sprengen, daher wird im Folgenden eine grobe Abschätzung der Kosten für das SFT vorgenommen und dann in Bezug zum RLHF gesetzt. Die Hauptfaktoren in der Kostenberechnung des Supervised Fine-Tuning sind die Parameteranzahl des LLMs und die Tokenanzahl des Trainingssatzes. Ein Parameter wird typischerweise in halber Gleitkommapräzision (FP16) gespeichert, was einem Speicherbedarf von 2 Byte pro Parameter entspricht [NVIDIA, 2018]. Darüber hinaus existieren weitere Ansätze zur Reduktion der benötigten Rechenleistung, aber oft auch der Effizienz, wie beispielsweise LoRA [Hu et al., 2021], auf die im Rahmen dieser Arbeit jedoch nicht näher eingegangen wird.

Eine für Modelltraining ausgelegte high-end GPU wie die NVIDIA A100-GPU erreicht ungefähr 3,12*10^{14} Floating-Point Operations Per Second (FLOPS) pro Sekunde im FP16 Modus [NVIDIA, 2020]. Es wird angenommen, die Kosten einer A100 GPU liegen bei 3,5 Euro pro Stunde, was ein realistischer Durchschnittspreis bei Cloudanbietern ist [GetDeploying, 2025].

Im Folgenden wird eine grobe Vergleichsrechnung für die Kosten des FP16 Fine-Tunings vom verwendeten LLM Gpt 4.1 sowie von neueren state-of-the-art Modellen mit jeweils 10 Millionen Trainingstoken durchgeführt. Auf Basis publizierter Modelle wie GPT-3 mit 175 Milliarden Parametern [Brown et al., 2020] und aktueller Übersichtsarbeiten zu Large Language Models [Zhao et al., 2023] erscheint es plausibel, für moderne state-of-the-art Modelle wie Gpt 5.1 , Gemini 3 oder Grok 5 von Parameteranzahlen im Bereich von mehreren Billionen auszugehen. Die Parameteranzahl von Gpt 4.1 ist ebenfalls nicht offiziell dokumentiert und wird hier auf eine Billionen geschätzt. Für die Berechnung wird die Formel $FLOPs \approx 6 \cdot N_{\text{Parameter}} \cdot N_{\text{Token}}$ benutzt [Kaplan et al., 2020]:

Kostenabschätzung FP16 für GPT 4.1 mit 10^{12} Parametern
FLOPs-Bedarf:

$$FLOPs \approx 6 \cdot 10^{12} \cdot 10^{7} = 6 \cdot 10^{19}$$

Rechenzeit:

$$\frac{FLOPs}{312 \cdot 10^{12}\,\text{FLOPs/s}} = \frac{6 \cdot 10^{19}}{312 \cdot 10^{12}} \approx 1{,}92 \cdot 10^{5}\,\text{s} \approx 53{,}3\,\text{Stunden}$$

Bei beispielhaften Kosten von 3,50 Euro pro GPU-Stunde ergäbe sich:

$$53{,}3\,\text{h} \cdot 3{,}50\,/\text{h} \approx 187\,\text{Euro}$$

Kostenabschätzung FP16 für modernes Modell mit $5 \cdot 10^{12}$ Parametern
FLOPs-Bedarf:

$$FLOPs \approx 6 \cdot 5 \cdot 10^{12} \cdot 10^{7} = 3 \cdot 10^{20}$$

Rechenzeit:

$$\frac{FLOPs}{312 \cdot 10^{12}\,\text{FLOPs/s}} = \frac{3 \cdot 10^{20}}{312 \cdot 10^{12}} \approx 9{,}62 \cdot 10^{5}\,\text{s} \approx 267{,}3\,\text{Stunden}$$

Bei denselben Cloudkosten ergibt sich:

$$267{,}3\,\text{h} \cdot 3{,}50\,/\text{h} \approx 936\,\text{Euro}$$

Die nötigen Rechenkosten für das Fine-Tuning mit einem Datensatz von 10 Millionen Trainingstokens liegen somit nicht im extrem hohen Bereich, sondern bewegen sich in einer Größenordnung, die für Unternehmen mit Cloud-Budget wirtschaftlich realisierbar ist. Die vorgenommenen Berechnungen erfassen ausschließlich die Rechenkosten für das SFT und berücksichtigen nicht den Aufwand für die Aufbereitung der Trainingsdaten. Bei der Durchführung von RLHF liegen die Kosten bei gleicher Anzahl an Trainingstokens mindestens in derselben Größenordnung. In der Praxis fallen durch manuelle Bewertungen zusätzlich Personalkosten an, sodass RLHF teurer ist als reines SFT. Insgesamt ist die Durchführung von RLHF damit finanziell und technisch anspruchsvoll und erfordert weitergehende Vorbereitungen, während folgende Weiterentwicklungen mit deutlich geringerem Aufwand realisierbar sind.

7.2 Skalierbarkeit

Außerdem könnten folgende Funktionen erweitert werden, um den Agenten auf mehr Aufgaben zu skalieren und ihn nutzerfreundlicher zu gestalten.

- Durch eine mögliche Erweiterung der Eingabe von weiteren Entitäten aus Fremdsystemen kann die Wiederverwendbarkeit des Mapping-Agenten über den spezifischen Usecase erhöht werden.
- Eine andere Skalierung stellt die Möglichkeit zur Spracheingabe von Feedback dar, sodass die Nutzung erleichtert wird.
- Außerdem kann im verwendeten Copilot durch zusätzliche Einstellungen die automatische Kontextsuche verbessert werden, was zu einer einfacheren Bedienung mit weniger notwendigen Prompteingaben führt.

7.3 Integration

Die Arbeit wurde mit dem Ziel der Schnittstellenanbindung für FLEX konzipiert, ein naheliegender nächster Schritt ist daher die Integration ins Laufzeitsystem. Zunächst müssen die generierten Skripte der Syntax der FLEX-REST-API entsprechen, was durch angepasste Skript-Templates umgesetzt werden kann, sodass die bereitgestellte API diese Skripte ausführen kann. Der Hauptaufwand der Integration liegt in der notwendigen Interaktion des Agenten mit dem System. Es ist eine Regelung erforderlich, die eingehende Dateien eindeutig den jeweils zuständigen Mappingskripten zuordnet. Bei unbekannten Typen muss der Agent die eingehende Datei erhalten und ein Initialmapping generieren. Der generierte Output wird im System abgelegt werden und es ist für Nutzer eine Umgebung bereitzustellen, in der sie Feedback mitteilen können. Dieses Feedback sollte zunächst verpflichtend sein, um Fehler zu minimieren. Grundsätzlich erfolgt die technische Integration dabei über Schnittstellen zwischen Agent und FLEX-Laufzeitsystem. Zudem werfen diese Anforderungen Datenschutzfragen auf, die das zugrunde liegende LLM erfüllen muss.

8 Diskussion

Im Rahmen der Arbeit wurden folgende Ziele erreicht. Der prototypische Ansatz zeigt, dass sich Mappingskripte auf Basis fester Vorgaben schnell und zuverlässig erzeugen lassen. Änderungen in zugrunde liegenden Regeln werden vom Agenten erkannt und resultieren in entsprechend angepassten Skripten, ohne dass die Logik manuell neu implementiert werden muss. Die Entwicklung verschiebt den Fokus von technischen Details der Schnittstellenimplementierung hin zur fachlichen Bedeutung der Mappings. Der Agent besitzt das Potenzial, Wissen zu vorhandenen Zusammenhängen zuverlässig auf neue Dateitypen anzuwenden. Durch Fine-Tuning kann das Initialmapping noch verbessert werden. Insgesamt erscheint damit auch eine zukünftige Integration in das FLEX-Laufzeitsystem prinzipiell realistisch.
Gleichzeitig ist die Generierungsfähigkeit des Agenten weitgehend auf vorhandenes Wissen und explizit vorgegebene Regeln beschränkt. In praktischen Anwendungsfällen wird häufig kundenspezifische Logik benötigt oder die Ziel-Feldnamen unterscheiden sich gänzlich von den entsprechenden Quell-Feldnamen, sodass die Beziehungen nicht ohne weiteres automatisch abgeleitet werden können. Daher ist eine menschliche Unterstützung erforderlich, die nicht ableitbare fachliche Logik beisteuert. Darüber hinaus besteht ein Konsistenzrisiko: leicht variierende Nutzereingaben führen nicht immer zu identischen Ergebnissen, weshalb standardisierte Prompts und feste Arbeitsabläufe erforderlich sind. Schließlich müssen in einer praktischen Umsetzung auch Integrationsaufwand und API-Kosten berücksichtigt werden.

9 Fazit

Durch eine präzise Beschreibung des Usecases sowie expliziten Vorgaben wurde der Agent zielgerichtet konfiguriert und liefert mit menschlicher Überwachung zuverlässige Ergebnisse.
Bei der Integration in ein reales System muss der entstehende Integrationsaufwand gegen die resultierende Effizienzsteigerung abgewogen werden.
Bei dynamischen Systemen wie FLEX ist ein Mehrwert zu erwarten, da Schnittstellen einen hohen Anteil am Entwicklungsaufwand haben und fortlaufend angepasst werden müssen. In diesem Kontext kann ein Mapping-Agent wiederkehrende Anpassungen unterstützen und so den Aufwand langfristig reduzieren.
Jedoch stellt die Lösung keine vollautomatisierte Schnittstellengenerierung dar, weil eine menschliche Komponente in der Validierungsschicht für eine höchstmögliche Sicherheit eingebunden werden muss.

A Anhang: Konkretes Beispiel-Mapping

```
<SHPMNT05>
  <IDOC BEGIN="1">
    <EDI_DC40 SEGMENT="1">
      <IDOCTYP>SHPMNT05</IDOCTYP>
      <!-- ... weitere Felder ... -->
    </EDI_DC40>
    <E1EDT20 SEGMENT="1">
      <TKNUM>BD26TONNER</TKNUM>
      <SHTYP>0001</SHTYP>
      <TPBEZ>WA4_DATUM_UHRZEIT</TPBEZ>
      <E1EDL20 SEGMENT="1">
        <VBELN>9102031925</VBELN>
        <LGNUM>102</LGNUM>
        <ROUTE>GL_LP</ROUTE>
        <!-- ... weitere Felder ... -->
        <E1EDL37 SEGMENT="1">
          <EXIDV>30101056</EXIDV>
          <VEGR1>PICK</VEGR1>
          <E1EDL44 SEGMENT="1">
            <VELIN>1</VELIN>
            <VBELN>9102031925</VBELN>
            <POSNR>1</POSNR>
            <EXIDV>30101056</EXIDV>
            <VEMNG>1.00000000000000</VEMNG>
            <VEMEH>ST</VEMEH>
            <MATNR>KB0001</MATNR>
            <CHARG>UN00042895</CHARG>
          </E1EDL44>
        </E1EDL37>
      </E1EDL20>
    </E1EDT20>
  </IDOC>
</SHPMNT05>
```

Listing 1: Auszug einer XML-Eingangsdatei vom IDOCTYP SHPMNT05

```
TargetField;SourcePath;Comment;DefaultValue
startPosition;E1EDL20.LGNUM;Alternativ NAME2 oder EXIDV/EXIDV2 je nach Use Case;DEFAULT
targetPosition;E1EDL20.ROUTE;Alternativ LGNUM oder EXIDV/EXIDV2 je nach Use Case;DEFAULT
transportType;E1EDL20.ROUTE;Route als Transporttyp, falls leer Default;DEFAULT
processConfiguration;;Kein Wert im XML, fester Default;DEFAULT
amount;;Kein Wert im XML, fester Default;DEFAULT
transportID;E1EDL37.EXIDV2|E1EDL37.EXIDV|E1EDL44.EXIDV;TransportID wie ScanID;
priority;;Standardprioritaet, kann durch VSBED=JIT ueberschrieben werden;
latestEnd;;Nicht direkt belegbar;
earliestStart;;Nicht direkt belegbar;
remark;;Nicht direkt belegbar;
scanId;E1EDL37.EXIDV2|E1EDL37.EXIDV|E1EDL44.EXIDV;Bevorzugt EXIDV2, sonst EXIDV, sonst EXIDV aus E1EDL44;
customValues.referenceNumber;E1EDL20.VBELN;Referenznummer aus VBELN;
```

Listing 2: Mappingregeln für IDOCTYP SHPMNT05 wurden anhand eines anderen XMLs desselben IDOCTYPs generiert. Die Regeln wurden durch eine Feedbackiteration präzisiert, konkret wurden die Regeln für customValues.referenceNumber und transportID manuell ergänzt.

```
    // Annahme: mappedJson ist das Ergebnis von xml2js.
       parseString(xml)
var edi = mappedJson.SHPMNT05.IDOC.EDI_DC40;
var e1edl20 = mappedJson.SHPMNT05.IDOC.E1EDT20.E1EDL20;
var e1edl37Arr = Array.isArray(e1edl20.E1EDL37) ? e1edl20.
   E1EDL37 : [e1edl20.E1EDL37];
// Hilfsfunktion fuer Default-Werte
function notNull(val, def) {
    return (val !== undefined && val !== null && val !== "") ?
       val : def;
}
// TODO: Hier die feldspezifische Mappinglogik einfuegen (
   startPosition, targetPosition, ...)
var entity = {
    startPosition: startPosition,
    targetPosition: targetPosition,
    transportType: transportType,
    processConfiguration: processConfiguration,
    amount: amount,
    priority: priority,
    latestEnd: latestEnd,
    earliestStart: earliestStart,
    remark: remark,
    scanId: scanId,
    customValues: customValues
};
return JSON.stringify(entity, null, 2);
```

Listing 3: Vorgegebenes Script-Template für XML-Format. Es erwartet, dass das XML bereits in JSON geparsed wurde. Die Umwandlung erfolgt in einer separaten Parsingschicht der Testumgebung

```
// SHPMNT05 Mapping Script (strict, only per mapping rules)
// Input: mappedJson = result of xml2js.parseString(xml)

function notNull(val, def) {
    return (val !== undefined && val !== null && val !== "") ?
       val : def;
}

// Defensive: always check for existence
var idoc = mappedJson?.SHPMNT05?.IDOC || {};
var e1edl20 = idoc.E1EDT20?.E1EDL20 || {};
var e1edl37Arr = Array.isArray(e1edl20.E1EDL37) ? e1edl20.
   E1EDL37 : (e1edl20.E1EDL37 ? [e1edl20.E1EDL37] : []);
var e1edl37 = e1edl37Arr.find(x => x?.EXIDV2) || e1edl37Arr[0]
   || {};
var e1edl44 = e1edl37?.E1EDL44 || {};

// --- Mapping strictly per CSV ---
var entity = {
    startPosition: notNull(e1edl20.LGNUM, "DEFAULT"),
    targetPosition: notNull(e1edl20.ROUTE, "DEFAULT"),
    transportID: notNull(e1edl37.EXIDV2, notNull(e1edl37.EXIDV,
       notNull(e1edl44.EXIDV, null))),
    transportType: notNull(e1edl20.ROUTE, "DEFAULT"),
    processConfiguration: "DEFAULT",
    amount: "DEFAULT",
    priority: null,
    latestEnd: null,
    earliestStart: null,
    remark: null,
    scanId: notNull(e1edl37.EXIDV2, notNull(e1edl37.EXIDV,
       notNull(e1edl44.EXIDV, null))),
    customValues: {
        referenceNumber: notNull(e1edl20.VBELN, null)
    }
};

return JSON.stringify(entity, null, 2);
```

Listing 4: Vom Agenten generiertes Mapping-Skript für SHPMNT05 aus Template und Mappingregeln

```
{
  "startPosition": "102",
  "targetPosition": "GL_LP",
  "transportType": "GL_LP",
  "processConfiguration": "DEFAULT",
  "amount": "DEFAULT",
  "priority": null,
  "latestEnd": null,
  "earliestStart": null,
  "remark": null,
  "scanId": "30101056",
  "transportID": "30101056",
  "customValues": {
    "referenceNumber": "9102031925"
  }
}
```

Listing 5: Resultierendes Entitätsbjekt nach Ausführung auf lokaler HTML-Seite: Das XML wurde zuvor in JSON geparsed und anschließend an das Skript übergeben. Nicht alle Felder der Zielentität werden aus dem XML belegt, die leeren Pflichfelder sind DEFAULT und die restlichen sind null.

Literatur

[Bommasani et al., 2021] Bommasani, R., Hudson, D. A., and Adeli, E. (2021). On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*. URL: https://arxiv.org/abs/2108.07258.

[Brown et al., 2020] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901. URL: https://arxiv.org/abs/2005.14165.

[Buss and Safari, 2025] Buss, C. and Safari, M. (2025). Towards scalable schema mapping using large language models. *arXiv preprint arXiv:2505.24716*. URL: https://arxiv.org/pdf/2505.24716, Accessed: 2025-12-09.

[Chatterjee, 2025] Chatterjee, S. (2025). The impact of prompt bloat on llm output quality. URL: https://home.mlops.community/public/blogs/the-impact-of-prompt-bloat-on-llm-output-quality, Accessed: 2025-12-11, Blogartikel, MLOps Community.

[Chen et al., 2021] Chen, M., Tworek, J., Jun, H., and Yuan, Q. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*. URL: https://arxiv.org/abs/2107.03374.

[Dai and Pan, 2023] Dai, J. and Pan, X. (2023). Safe rlhf: Safe reinforcement learning from human feedback. URL: https://arxiv.org/pdf/2310.12773.

[Devlin et al., 2019] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*. URL: https://arxiv.org/abs/1810.04805.

[Fagin, 2009] Fagin, R. (2009). Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications*. Springer. URL: https://link.springer.com/chapter/10.1007/978-3-642-02463-4$_1$2.

[Gao et al., 2025] Gao, M., Li, Y., Liu, B., Yu, Y., Wang, P., Lin, C.-Y., and Lai, F. (2025). Single-agent or multi-agent systems? why not both? *arXiv preprint arXiv:2505.18286*. URL: https://arxiv.org/abs/2505.18286.

[Gao et al., 2023] Gao, Y., Zhang, H., Han, X., Liu, Z., and Sun, M. (2023). Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2305.18657*. URL: https://arxiv.org/abs/2305.18657, Accessed: 2025-12-09.

[GeeksforGeeks, 2023] GeeksforGeeks (2023). Tokenization in natural language processing. URL: https://www.geeksforgeeks.org/nlp/tokenization-in-natural-language-processing-nlp/.

[GetDeploying, 2025] GetDeploying (2025). Nvidia a100 gpu pricing and specifications. URL: https://getdeploying.com/gpus/nvidia-a100.

[Holtzman et al., 2020] Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. (2020). The curious case of neural text degeneration. In *International Conference on Learning Representations (ICLR)*. URL: https://arxiv.org/abs/1904.09751.

[Hu et al., 2021] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., and Wang, S. (2021). Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*. URL: https://arxiv.org/abs/2106.09685, Accessed: 2025-12-09.

[Hödl, 2025] Hödl, J. (2025). Generative künstliche intelligenz – individualisierte anpassung durch prompt engineering. URL: https://opus.campus02.at/files/1213/AC17639779.pdf, Accessed: 2025-12-11.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285. URL: https://www.jair.org/index.php/jair/article/view/10166/24110.

[Kaplan et al., 2020] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*. URL: https://arxiv.org/abs/2001.08361.

[Lewis et al., 2020] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*. URL: https://arxiv.org/abs/2005.11401.

[Li, 2024] Li, K. (2024). Measuring and controlling instruction (in)stability in language model dialogs. URL: https://arxiv.org/abs/2402.10962, Accessed: 2025-12-11.

[Liu et al., 2023] Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35. URL: https://arxiv.org/abs/2107.13586.

[Ma et al., 2025] Ma, L., Zhang, R., Han, Y., Yu, S., Wang, Z., Ning, Z., Zhang, J., and Xu, P. (2025). A comprehensive survey on vector database: Storage and retrieval techniques. *arXiv preprint arXiv:2310.11703*. URL: https://arxiv.org/html/2310.11703v2.

[Müller and Schneider, 2025] Müller, T. and Schneider, A. (2025). Next-token prediction in large language models: Probabilistic foundations. *Der Radiologe*, 65(12):1010–1020. URL: https://link.springer.com/article/10.1007/s00117-025-01427-z.

[Neubauer, 2025] Neubauer, F. (2025). Ai-assisted json schema creation and mapping. *arXiv preprint arXiv:2508.05192*. URL: https://www.arxiv.org/pdf/2508.05192, Accessed: 2025-12-09.

[NVIDIA, 2018] NVIDIA (2018). Mixed precision training. URL: https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html.

[NVIDIA, 2020] NVIDIA (2020). Nvidia a100 tensor core gpu architecture. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf.

[OpenAI, 2023] OpenAI (2023). Fine-tuning language models. URL: https://platform.openai.com/docs/guides/fine-tuning.

[Parciak et al., 2024] Parciak, M., Vandervoort, B., and Neeven, F. (2024). Schema matching with large language models: An experimental study. *arXiv preprint arXiv:2407.11852*. URL: https://arxiv.org/abs/2407.11852.

[Parnas, 1972] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058. URL: https://dl.acm.org/doi/10.1145/361598.361623.

[Reynolds and McDonell, 2021] Reynolds, L. and McDonell, K. (2021). Prompt programming for large language models: Beyond the few-shot paradigm. *arXiv preprint arXiv:2102.07350*. URL: https://arxiv.org/abs/2102.07350.

[Shapkin et al., 2023] Shapkin, A., Litvinov, D., Zharov, Y., Bogomolov, E., and Galimzyanov, T. (2023). Dynamic retrieval-augmented generation. *arXiv preprint arXiv:2312.08976*. URL: https://arxiv.org/abs/2312.08976.

[Tong and Zhang, 2024] Tong, W. and Zhang, T. (2024). Codejudge: Evaluating code generation with large language models. *arXiv preprint arXiv:2410.02184*. URL: https://arxiv.org/pdf/2410.02184, Accessed: 2025-12-09.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*. URL: https://arxiv.org/abs/1706.03762.

[von Richthofen et al., 2022] von Richthofen, A., Ogolla, F. O., and Send, H. (2022). How artificial intelligence affects knowledge work: A qualitative study of ai adoption in organizations. *Information*, 13(4):199. URL: https://www.mdpi.com/2078-2489/13/4/199.

[White et al., 2023] White, J. et al. (2023). A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv*. URL: https://arxiv.org/abs/2302.11382.

[Zhao et al., 2023] Zhao, W. X., Liu, J., et al. (2023). A survey of large language models. *arXiv preprint arXiv:2303.18223*. URL: https://arxiv.org/abs/2303.18223.