

KI-gestütztes Reverse Requirements Engineering bei Legacy-Software

Masterarbeit an der Hochschule Neu-Ulm

Masterarbeit

Master of Business Administration

Autor: Christoph Schwörer

Betreuer: Prof. Dr. Daniel Schallmö

Abgabedatum: XX 2026

University of Applied Sciences Neu-Ulm

Eigenständigkeitserklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Alle wörtlich oder sinngemäß übernommenen Textstellen sind als solche gekennzeichnet.

Neu-Ulm, _____

Unterschrift: _____

Abstract

Zusammenfassung der Arbeit.

Inhaltsverzeichnis

Eigenständigkeitserklärung	2
Abstract	3
1 Einleitung	2
1.1 Ausgangssituation und Motivation	2
1.2 Problemstellung	2
1.3 Zielsetzung	3
1.4 Forschungsleitfragen	3
1.5 Aufbau der Arbeit	3
2 Theoretische Grundlagen	4
2.1 Requirements Engineering und Reverse Requirements Engineering	4
2.2 Large Language Models im Software Engineering	9
2.3 Legacy-Modernisierung und Stand der Forschung	15
3 Fallstudie c-entron GmbH	18
3.1 Unternehmenskontext und Legacy-Software	18
3.2 Migrationsstrategie und spezifische Herausforderungen	19
4 Konzeption und methodisches Vorgehen	22
4.1 Werkzeug-Grundlagen: Agentische CLIs, Agenten, MCP und lokale Inferenz	22
4.2 Methodisches Design im Überblick	23
4.3 Der RRE-Prozess als Rahmen	24
4.4 Auswahl des LLM	25
4.5 Versuchsaufbau	25
4.6 Stakeholder-Validierung als Verifikationsverfahren	28
4.7 Evaluationsrahmen	29
4.8 Reproduzierbarkeit und Risikomanagement	29
4.9 Konkrete Konfigurationen der geplanten Versuche	30
5 Durchführung und Ergebnisse	32
6 Evaluation (ca. 12 Seiten)	33
7 Diskussion	33
7.1 Interpretation der Ergebnisse	33
7.2 Chancen und Grenzen	33
7.3 Implikationen fuer Forschung und Praxis	33
8 Fazit und Ausblick	33
9 Literaturverzeichnis (ca. 3 Seiten)	33
Bibliografie	33
10 Anhang (ca. 6 Seiten)	35
10.1 Interviewleitfäden	35
10.2 Zusätzliches Datenmaterial	35
10.3 Konfigurationsdetails des Prototyps	35

1 Einleitung

1.1 Ausgangssituation und Motivation

In den vergangenen Jahren hat die digitale Transformation mittelständische Softwareanbieter gezwungen, ihre Produkte neu zu bewerten. Betroffen sind vor allem Systeme, die über lange Jahre ausschließlich in Windows-Umgebungen vertrieben wurden. Diese stoßen bei Cloud-, Web- und Mobile-Szenarien an technische sowie organisatorische Grenzen und häufen zunehmend technische Schulden an. Eine technologische Weiterentwicklung wird zusehends schwieriger, und an einer Neuentwicklung führt oft kein Weg vorbei. Dokumentierte Anforderungen oder eine systematische Code-Dokumentation sind allerdings selten. Der Großteil des Wissens steckt implizit im Code oder in den Köpfen der verbleibenden Entwickler.

Die c-entron GmbH in Ulm ist von diesem Szenario unmittelbar betroffen. Das Unternehmen betreibt seit über zwanzig Jahren eine Windows-basierte ERP-Suite für IT-Systemhäuser. Die Lösung deckt Auftragsabwicklung, Lager, Fakturierung und Projektabrechnung ab, ist jedoch eng mit der bisherigen Client/Server-Architektur gekoppelt. Kunden erwarten inzwischen plattformunabhängige Oberflächen, Self-Service-Funktionen und flexible Betriebsmodelle wie SaaS (Software as a Service). Die bestehende Anwendung ist in Skalierung, Deployment und Abrechnung jedoch limitiert, sodass eine Migration auf eine webbasierte Plattform zwingend erforderlich wird.

Eine Neuimplementierung auf Basis vorhandener Anforderungs- oder Code-Dokumentation ist aus den oben genannten Gründen kaum möglich. Die Herausforderung besteht darin, mit möglichst geringem Aufwand eine vollständige Beschreibung eines komplexen ERP-Systems zu erarbeiten. Eine manuelle Auswertung von Code und Oberflächen auf Funktionalitäten ist angesichts der Komplexität, die aus der langjährigen Weiterentwicklung resultiert, nur mit sehr hohem Personalaufwand möglich und daher nicht realisierbar.

In den vergangenen Jahren hat sich hierzu ein neues Instrumentarium etabliert. Large Language Models wie ChatGPT-5 oder Claude.ai können in Verbindung mit agentischen CLIs (Codex, Claude Code) große Mengen an Quellcode analysieren, Anforderungen erarbeiten und textuell beschreiben. Damit entsteht die Chance, fehlende Anforderungsdokumentationen zumindest teilweise aus dem Code heraus zu rekonstruieren. Die praktische Nutzung dieses Potenzials ist bislang kaum erforscht. Diese Arbeit setzt an dieser Stelle an und untersucht, wie KI-gestützte Verfahren für eine systematische Anforderungsanalyse eingesetzt werden können.

1.2 Problemstellung

Für das ERP-Software-Produkt der c-entron fehlen strukturierte und dokumentierte Requirements. Die Analyse der bestehenden Codebasis ist zeit- und ressourcenintensiv sowie anfällig für Insel- und Metawissen. Daraus ergeben sich mehrere Risiken:

Re-Implementationsfehler

Edge Cases, Workarounds und kundenindividuelle Anpassungen sind nur im Code sichtbar. Ohne vollständige Erfassung drohen Funktionsverluste nach der Migration. Workarounds sind zugleich Symptom einer mangelhaften Anforderungserfassung in der ursprünglichen Implementierung und Zeichen fehlender Weitsicht.

Technische Schuld

Entwickler investieren viel Zeit in das Verständnis historischer Strukturen, statt aktiv an der neuen Plattform zu arbeiten. Veraltete Muster werden unreflektiert übernommen. Neue Mitarbeiter sind mit den alten Technologien nicht vertraut, und es fehlt ihnen das Verständnis für historische Zwänge und Zusammenhänge.

Implizites Wissen

Domänenwissen liegt bei wenigen langjährigen Mitarbeitenden. Personalwechsel führen zu Wissensverlust und Verzögerungen. Gleichzeitig führt die langjährige Arbeit mit dem bestehenden System zu eingeschränkter Offenheit beim Design neuer Lösungen („Das haben wir schon immer so umgesetzt,,).

Komplexität der Codebasis

Verschachtelte Abhängigkeiten, unterschiedliche Stile und technologiebedingte Zwänge erschweren eine modulare Anforderungsableitung.

Fehlende Traceability

Ohne Zuordnung zwischen Code und Geschäftsprozess fehlt die Grundlage für Priorisierung, Testkonzeption und spätere Wartung. Große Teile des Codes sind zudem generisch und lassen sich keiner konkreten Anforderung zuordnen, etwa das Anzeigen einer Tabelle. Konkrete Datenflüsse lassen sich nur am laufenden System beobachten, was die Analyse um eine weitere Größenordnung ressourcenintensiver macht.

Eine rein manuelle Rekonstruktion aller Anforderungen wäre wirtschaftlich kaum tragbar. Deshalb soll geprüft werden, ob KI-gestützte Verfahren Requirements so extrahieren können, dass sie als belastbare Basis für die Modernisierung dienen.

1.3 Zielsetzung

Diese Arbeit verfolgt das Ziel, ein vollständiges Vorgehen für KI-gestütztes Reverse Requirements Engineering im Umfeld eines mittelständischen ERP-Herstellers zu entwickeln und zu bewerten. Die Teilziele lauten:

- Entwicklung eines Prozessmodells, das Vorbereitung, Analyse, Validierung und Übergabe strukturiert.
- Evaluation aktueller LLMs hinsichtlich Kontextfenster, Codeverständnis, Steuerbarkeit, Kosten und Datenschutz.
- Definition eines Evaluationsrahmens mit quantitativen und qualitativen Kriterien (Vollständigkeit, Verständlichkeit, Redundanzfreiheit, Aufwandseinsparung).
- Integration von Stakeholder-Wissen durch Interviews, um die Qualität der KI-Ergebnisse zu bewerten und nicht direkt aus dem Code ableitbare Anforderungen zu ergänzen.

1.4 Forschungsleitfragen

Diese Zielsetzung wird über vier Forschungsleitfragen strukturiert:

- a) **Einsatz von LLMs im Reverse Requirements Engineering:** Welche Prozessschritte, Steuerungsmechanismen und Kontrollpunkte sind notwendig, um LLMs reproduzierbar einzusetzen?
- b) **Kombination von KI-Analyse und Stakeholder-Input:** Welche funktionalen und nicht-funktionalen Anforderungen lassen sich aus Code extrahieren, und welche Informationen müssen über Interviews ergänzt werden?
- c) **Qualitätsbewertung der generierten Requirements:** Wie beurteilen Fachexperten Vollständigkeit, Verständlichkeit, Nützlichkeit und Aufwandseinsparung der KI-Ergebnisse?
- d) **Chancen und Grenzen des Ansatzes:** Welche Effizienzgewinne sind realistisch, wo liegen technische oder organisatorische Limitierungen, und welche Risiken (z. B. Halluzinationen, Datenschutz) müssen adressiert werden?

1.5 Aufbau der Arbeit

Die Arbeit ist in acht Kapitel gegliedert:

- a) **Einleitung:** Kontext, Problemstellung, Ziele und Forschungsfragen.
- b) **Theoretische Grundlagen:** Requirements Engineering, Reverse Engineering, Large Language Models sowie Qualitätssicherungskriterien.

- c) **Fallstudie c-entron GmbH:** Unternehmensprofil, Produktarchitektur, Migrationsdruck und Rahmenbedingungen.
- d) **Konzeption und methodisches Vorgehen:** Prozessmodell, Technologieauswahl, Stakeholder-Einbindung und Datenbasis.
- e) **Ergebnisse:** Vollständige Ergebnisdarstellung der drei Versuche inkl. Artefaktlisten und beispielhafter Requirements / Use Cases aus den Ergebnisverzeichnissen.
- f) **Evaluation:** Vorgehen, Metriken, Ergebnisse und Expertenfeedback.
- g) **Diskussion:** Interpretation der Resultate, Limitationen und Implikationen für Forschung und Praxis.
- h) **Fazit und Ausblick:** Zusammenfassung, Beantwortung der Forschungsfragen und Perspektiven für weitere Arbeiten.

2 Theoretische Grundlagen

Dieses Kapitel beschreibt die theoretischen Grundlagen, die für die Konzeption und Bewertung eines KI-gestützten Reverse Requirements Engineering in Legacy-Umgebungen benötigt werden. Zunächst werden zentrale Themen des Requirements Engineerings sowie die Idee des reverse Requirements Engineerings auf Basis bestehender Systeme eingeordnet. Anschließend werden Large Language Models und deren Einsatz im Software Engineering inklusive typischer Leistungsgrenzen und Absicherungsmechanismen beschrieben. Abschließend werden Grundlagen der Legacy-Modernisierung sowie etablierte Migrationsstrategien zusammengefasst, um den Kontext der Fallstudie und die Zielrichtung einzuordnen.

2.1 Requirements Engineering und Reverse Requirements Engineering

2.1.1 Begriff und Zielsetzung des Requirements Engineering

Der Begriff Requirements Engineering (RE) umfasst die systematische Erhebung, Analyse, Spezifikation, Validierung und Verwaltung von Anforderungen an ein System über dessen Lebenszyklus. In Standards (IEEE, 1998; ISO/IEC/IEEE, 2018) wird Requirements Engineering als eigenständiger Prozess verstanden, der sowohl fachliche Ziele (z. B. unterstützte Geschäftsprozesse) als auch technische und organisatorische Randbedingungen (z. B. Sicherheitsvorgaben, Betriebsmodelle) in überprüfbare Aussagen überführt.

Im Kern adressiert das Requirements Engineering zwei Themen:

Kommunikation zwischen Domäne und Technik

Anforderungen müssen fachlich verständlich und gleichzeitig so präzise sein, dass sich daraus eine Software-Architektur ableiten lässt, die implementiert, getestet und geändert werden kann.

Umgang mit Unsicherheit und Wandel

Anforderungen sind zu Projektbeginn selten vollständig. Requirements Engineering ist daher nicht nur Dokumentation, sondern auch ein iterativer Klärungs- und Abstimmungsprozess.

Ein etablierter Ansatz zur Strukturierung von diversen Sichtweisen ist das Viewpoint-Konzept (Kotonya & Sommerville, 1996), bei dem Anforderungen aus unterschiedlichen Perspektiven modelliert und anschließend konsolidiert werden.

Für diese Arbeit ist die Perspektivenorientierung relevant, weil implementierter Code typischerweise keine expliziten Stakeholder-Sichten enthält. Für eine Migration auf Basis eines Reverse-Engineering-Ansatzes sind diese aber relevant für die Implementierung und architekturelle Entscheidungen (z. B. Nutzerrollen, kundenspezifische Varianten, regulatorische Vorgaben).

2.1.2 Arten von Requirements und Qualitätskriterien

In der Literatur wird häufig zwischen funktionalen Anforderungen (Was soll das System tun?) und Qualitäts- bzw. nicht-funktionalen Anforderungen (Welche Eigenschaften und Randbedingungen gelten?) unterschieden. Die Praxis zeigt jedoch, dass diese Trennung nicht immer scharf ist. Eigenschaften können sowohl als Systemverhalten (z. B. „Audit-Log erzeugen,“) als auch als Qualitätsziel (z. B. „Nachvollziehbarkeit“) formuliert werden (Glinz, 2007). Für das Reverse Requirements Engineering ist diese Unschärfe besonders relevant, weil Quellcode meist Verhalten konkretisiert, Qualitätsziele aber häufig implizit bleiben (z. B. Performance-Workarounds, Sicherheitsannahmen).

Für die Qualität einzelner Requirements gibt es etablierte Standards. (ISO/IEC/IEEE, 2018) nennt unter anderem Eindeutigkeit, Konsistenz, Vollständigkeit, Verifizierbarkeit und Nachvollziehbarkeit als zentrale Eigenschaften. (IEEE, 1998) formuliert ähnliche Prinzipien für Software Requirements Specifications, mit stärkerem Fokus auf Dokumentstruktur und Lesbarkeit.

Für die Bewertung von KI-extrahierten Requirements sind drei Kriterien maßgeblich relevant:

Verifizierbarkeit

Ein Requirement ist so formuliert, dass ein Test oder eine Prüfmethode ableitbar ist (z. B. Messkriterium, Akzeptanzbedingung).

Eindeutigkeit

Formulierungen vermeiden Mehrdeutigkeiten und definieren Begriffe, die in der Domäne unterschiedlich interpretiert werden können

(z.B. „Das System soll Aufträge schnell verarbeiten,“ vs. „Das System soll einen Auftrag innerhalb von 2 Sekunden validieren und bestätigen“)

Nachvollziehbarkeit (Traceability)

Es ist erkennbar, aus welchem Requirement das Artefakt (Code, Konfiguration, Datenbank, Ticket, Interview) abgeleitet wurde.

Nicht funktionale Anforderungen (z.B. Qualitätsanforderungen) bedürfen einer besonderen Betrachtung, weil sie über die reine Funktionsgleichheit hinaus die Zielarchitektur bestimmen. Glinz (2008) argumentiert, dass Qualitätsanforderungen risikobasiert und wertorientiert priorisiert werden sollten. Für Legacy-Migrationen ist dies nachvollziehbar: Ein „vollständiges,“ Requirements-Set ist praktisch schwer erreichbar, gleichzeitig sind bestimmte Non-Functional Requirements (z. B. Datenschutz, Verfügbarkeit, Rollout-Fähigkeit) hochkritisch, weil sie Architekturentscheidungen dominieren.

Für die inhaltliche Strukturierung von Qualitätsanforderungen ist das Qualitätsmodell ISO/IEC 25010:2011 verbreitet, das Qualitätsmerkmale wie Performance-Effizienz, Zuverlässigkeit, Sicherheit oder Wartbarkeit systematisch ordnet. Für Reverse Requirements Engineering ist dies hilfreich, weil aus Code häufig nur Teilaspekte sichtbar werden (z. B. Caching-Mechanismen als Hinweis auf Performance-Annahmen), während andere Qualitätsziele (z. B. „Maintainability,“) eher indirekt über Architekturentscheidungen und Entwicklungspraktiken wirksam werden (ISO/IEC, 2011).

Die Relevanz sauber formulierter Requirements zeigt sich auch in der Risikoperspektive. B et al. (2001) beschreiben Requirements Engineering als primäres Risiko, wenn Anforderungen unklar, instabil oder unvollständig sind.

Für diese Arbeit folgt daraus, dass KI-gestütztes Reverse-Requirements-Engineering nicht nur „mehr Text,“ erzeugen darf, sondern gezielt die Risiken der Unklarheit und der Fehlinterpretation reduzieren muss.

2.1.3 Spezifikationsformen und Grad der Formalisierung

Requirements werden in unterschiedlichen Repräsentationsformen dokumentiert. Standards wie IEEE 830-1998 und ISO/IEC/IEEE 29148:2018 fokussieren auf strukturierte Spezifikationen (z. B. SRS) und definieren typische Kapitel (Zweck, Systemkontext, funktionale Anforderungen, Schnittstellen, Qualitätsanforderungen, Annahmen). Zudem existieren weniger formale Formen wie User Stories, Use-Case-Beschreibungen oder Backlog-Einträge, die in agilen Settings Verwendung finden (IEEE, 1998; ISO/IEC/IEEE, 2018).

Für Reverse Requirements Engineering sind zwei Punkte entscheidend:

- **Form** beeinflusst Interpretierbarkeit: Eine kurze User Story („Als Nutzer möchte ich ...“) ist leicht verständlich, transportiert aber weniger Randbedingungen, Datenregeln oder Fehlerfälle. Eine SRS-Formulierung kann präziser sein, erfordert aber mehr Kontext und Definitionen.
- **Grad** der Formalisierung beeinflusst Prüfbarkeit: Je stärker Requirements mit Akzeptanzkriterien, Beispielen oder Messgrößen verknüpft sind, desto einfacher sind Reviews und Tests. Pohl (2010) betont Anforderungen-Validierung als eigene Disziplin.

Für diese Arbeit wird daher ein hybrider Stil gewählt: Requirements werden als kurze, klare Soll-Aussagen formuliert und jeweils um Kontext (Akteur/Prozess), Randbedingungen (Vorbedingungen, Datenobjekte) und mindestens eine Prüffidee ergänzt.

2.1.4 Traceability als Verbindung zwischen Code und Requirement

Traceability bezeichnet die Verknüpfung von Requirements und Artefakten, wie z.B. Code oder Test. Gotel & Finkelstein (o. J.) bezeichnen Traceability als wiederkehrendes Problem, vor allem bei unterschiedlichen Arten von Artefakten.

Beim Reverse Requirements Engineering ist Traceability nicht nur ein „Nice-to-have,, sondern eine Grundvoraussetzung.

Ein Requirement lässt sich gegen konkrete Codeausschnitte oder Laufzeitbeobachtungen prüfen, da das Requirement ja aus dem Code selbst entsteht.

In Legacy-Systemen ist die ursprüngliche Traceability typischerweise unvollständig, falls überhaupt vorhanden: Hinweise finden sich in Commit-Messages, Branch-Namen, Datenbankskripten, Konfigurationsdateien, UI-Texten oder in impliziten Konventionen. Der Anspruch dieser Arbeit besteht daher nicht darin, „perfekte,, Traceability wiederherzustellen, sondern eine minimal belastbare, reproduzierbare Verknüpfung zwischen extrahierten Requirements und Artefakten zu etablieren.

2.1.5 Abgrenzung von Reverse Engineering zu Reverse Requirements Engineering

Reverse Engineering wird klassisch als Analyseprozess verstanden, der aus einem bestehenden System Wissen über Struktur, Verhalten und Designentscheidungen rekonstruiert. Chikofsky & Cross (1990) prägen hierfür die Benennung und grenzen Reverse Engineering von Reengineering sowie Design Recovery ab. Für Requirements-nahe Fragestellungen ist hier relevant, dass Reverse Engineering nicht automatisch auch Requirements liefert, sondern erstmal nur technische Fakten (z. B. Abhängigkeiten, Datenflüsse, Zustandsautomaten).

Reverse Requirements Engineering (RRE) fokussiert sich dagegen auf die rückwärtsgerichtete Gewinnung von Requirements aus bestehenden Artefakten. Dabei kann das Ziel unterschiedlich interpretiert werden:

Rekonstruktion eines Soll-Zustands

Welche fachlichen Anforderungen werden durch die aktuelle Implementierung implizit erfüllt? Was war das ursprüngliche Ziel der Implementierung?

Rekonstruktion eines Ist-Zustands

Welche Funktionen und Regeln sind dagegen tatsächlich implementiert?

Gerade im Legacy-Umfeld ist diese Unterscheidung entscheidend. Die Codebasis enthält oft historisch entstandene Workarounds oder kundenspezifische Anpassungen. Ohne zusätzliche Validierung besteht das Risiko, dass RRE den Ist-Zustand als Soll-Zustand fehlinterpretiert.

Frühe Ansätze zur Brücke zwischen Reverse Engineering und Requirements liefern beispielsweise Yu et al. (o. J.) mit „RETR: Reverse Engineering to Requirements,. Der Beitrag betont, dass Requirements-Rückgewinnung eine methodische Kette aus Artefaktsichtung, Strukturierung und Validierung benötigt.

Methodisch lassen sich dabei grob zwei Analysestränge unterscheiden:

Statische Analyse

Ableitung von Struktur- und Datenflussinformationen aus Code und Artefakten ohne Ausführung (z. B. Abhängigkeiten, SQL-Statements, Aufrufketten). Statische Analyse skaliert gut, erkennt aber nicht zuverlässig Laufzeitbedingungen (z. B. Feature Flags, Konfigurationsvarianten).

Dynamische Analyse

Beobachtung von Laufzeitverhalten durch Logging, Tracing oder instrumentierte Tests (z. B. welche Regeln bei bestimmten Eingaben greifen). Dynamische Analyse ist näher am realen Verhalten, benötigt aber reproduzierbare Szenarien und Testdaten.

Reverse Requirements Engineering in einem Migrationsprojekt profitiert typischerweise von einer Kombination beider Stränge. Ohne dynamische Belege steigt das Risiko, dass nicht offensichtliche Bedingungen (z. B. kundenspezifische Schalter) übersehen werden; ohne statische Analyse bleibt die Abdeckung häufig zu gering.

Da eine Dynamische Analyse durch ein LLM mit den gegebenen technischen Möglichkeiten derzeit unpraktikabel ist, fokussiert sich diese Arbeit auf die statische Analyse von Artefakten, ergänzt manuell erstellte Artefakte zur Laufzeit (z.B. Screenshots).

2.1.6 Typische Methodenkette für Requirements-Rückgewinnung aus Code

Aus Sicht dieser Arbeit lässt sich Reverse Requirements Engineering einer Legacy-Codebasis als wiederholbarer Ablauf darstellen. Die konkrete Implementierung hängt vom System und den verfügbaren Artefakten ab, die grundlegenden Schritte sind jedoch weitgehend stabil:

- a) **Scope und Domänenabgrenzung:** Auswahl relevanter Module, Datenobjekte und Prozesse (z. B. Auftragsabwicklung, Fakturierung).
- b) **Artefakterhebung:** Quellcode, Konfiguration, UI-Texte, Datenbankschemata, Schnittstellenbeschreibungen, Change-Historie.
- c) **Technische Analyse:** Struktur- und Abhängigkeitsanalyse, Identifikation von Kernkomponenten, Regeln und Integrationspunkten.
- d) **Semantische Interpretation:** Ableitung fachlicher Aussagen aus technischen Implementierungen (z. B. Statusübergänge, Berechtigungsprüfungen).
- e) **Formalisierung als Requirements:** Überführung in klare, testbare Anforderungen mit Kontext (Akteur, Vorbedingung, Ergebnis).
- f) **Traceability-Anreicherung:** Verknüpfung jedes Requirements mit Belegen (Datei, Klasse, Methode, SQL-Statement, UI-String).
- g) **Validierung:** Review durch Fachexperten und Abgleich mit Laufzeitverhalten, Tickets oder Kundenwissen.

In der Praxis unterscheiden sich Artefakte darin, wie direkt sie fachliche Aussagen stützen. Quellcode, der eine Regel hart erzwingt (z. B. „Update nur bei Status X,“), ist als Beleg stärker als Kommentare

oder UI-Texte, die lediglich Absichten ausdrücken. Für eine belastbare Requirementsbasis ist es daher sinnvoll, Belege zu klassifizieren und die Aussagekraft zu kennzeichnen, beispielsweise:

Primärbelege

Durchgesetzte Regeln im Code oder in Datenbankconstraints (z. B. Statusmaschinen, Validierungslogik, Berechtigungschecks).

Sekundärbelege

Indirekte Hinweise wie UI-Labels, Fehlermeldungen, Report-Layouts, Mappingtabellen oder Konfigurationsschalter.

Kontextbelege

Ticketbeschreibungen, Commit-Messages oder Interviewaussagen, die Motivation und Ausnahmen erklären, aber nicht zwingend im Code sichtbar sind.

Diese Einteilung dient der Risikobewertung: Requirements, die überwiegend auf Sekundär- oder Kontextbelegen beruhen, sind anfälliger für Fehlinterpretation und sollten priorisiert validiert werden. Datenbankschemata und SQL-Statements sind häufig besonders aussagekräftig, weil sie Domänenobjekte, Kardinalitäten und Geschäftsregeln (z. B. referentielle Integrität, historisierte Tabellen) abbilden.

Für diese Arbeit werden in der Schrittfolge lediglich Punkte 1 und 7 manuell durchgeführt, während die Schritte 2-6 durch KI-gestützte Analyse automatisiert werden sollen.

2.2 Large Language Models im Software Engineering

2.2.1 Künstliche Intelligenz, Machine Learning und Einordnung von LLMs

Die Einordnung von Large Language Models folgt einer hierarchischen Begriffsstruktur, die sich in vier Ebenen gliedern lässt (vgl. Abbildung 1):

Künstliche Intelligenz (KI) ist der Oberbegriff für Verfahren, die Aufgaben bearbeiten, welche in der Praxis typischerweise kognitive Fähigkeiten erfordern (z. B. Klassifikation, Planung, Sprachverarbeitung) (Bishop, 2006).

Machine Learning (ML) ist ein Teilgebiet der KI, das Modelle aus Daten lernt, anstatt Regeln vollständig manuell zu spezifizieren (sogenannte Expertensysteme). In der Praxis wird zwischen Systemen mit überwachtem Lernen (mit Zielwerten), unüberwachtem Lernen (ohne Zielwerte, System erkennt selbst eine Struktur) und Reinforcement Learning (Lernen über Rückmeldung, z.B. Ergebnis Richtig / Falsch) unterschieden (Bishop, 2006; Goodfellow et al., 2016). In der heutigen Welt sind Neuronale Netze die dominierende ML-Architektur, insbesondere für unstrukturierte Daten wie Text und Code.

Deep Learning ist wiederum ein Teilbereich von ML, der neuronale Netze mit vielen Parametern und vielen (tiefen) Verarbeitungsebenen nutzt, um geeignete Repräsentationen aus Rohdaten zu lernen. Charakteristisch ist, dass Merkmalsextraktion und Modellanpassung gemeinsam über Optimierung (typischerweise Gradientenverfahren) erfolgen.

Large Language Models (LLMs) sind eine spezielle Ausprägung von Neuronalen Netzen, die auf sehr großen Text- und Codemengen vortrainiert werden. Zudem verfügen sie über sehr große Mengen (>Milliarden) von Eingabeparametern (Tokens) und viele Schichten (>100). In der aktuellen Modellgeneration dominiert die Transformer-Architektur (Vaswani et al., 2017), deren Funktionsweise in den folgenden Abschnitten erläutert wird.

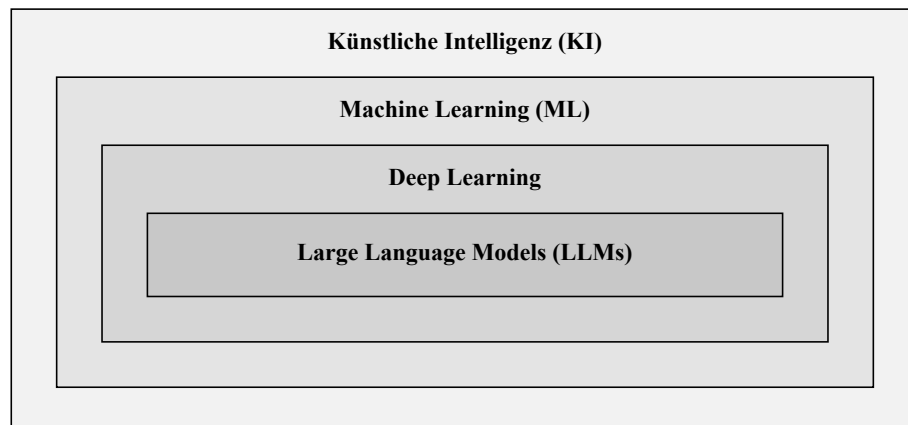


Abbildung 1: Hierarchische Einordnung: KI $\supset$ ML $\supset$ Deep Learning $\supset$ LLMs.

Neuronale Netze lassen sich vereinfacht als parametrisierte Funktionsketten aus Schichten (Layers) beschreiben. Dabei geben die Schichten die Eingaben in zunehmend abstrakte Repräsentationen jeweils in die nächste Schicht weiter. Das Training erfolgt über eine Zielfunktion und Gradientenberechnung. In der Praxis geschieht dies meist über Backpropagation und Varianten des Gradientenabstiegs (Goodfellow et al., 2016).

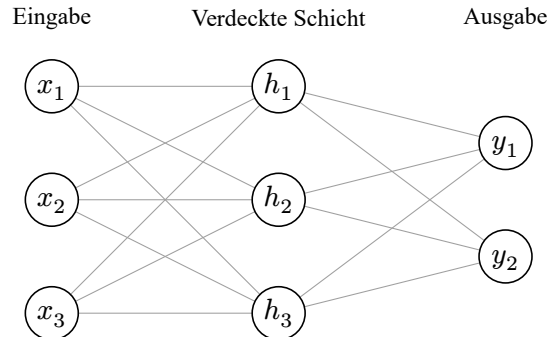


Abbildung 2: Schematische Darstellung eines vollständig verbundenen Feedforward-Netzes.

Ein einzelnes Neuron lässt sich als Transformation mit nachgeschalteter Aktivierungsfunktion formulieren:

$$z = \sum_{i=1}^d w_i x_i + b, \quad a = \varphi(z)$$

Typische Aktivierungsfunktionen ($a = \varphi(z)$) sind die Sigmoid-Funktion, der hyperbolische Tangens und ReLU (Goodfellow et al., 2016):

Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

tanh

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

ReLU

$$\text{ReLU}(z) = \max(0, z)$$

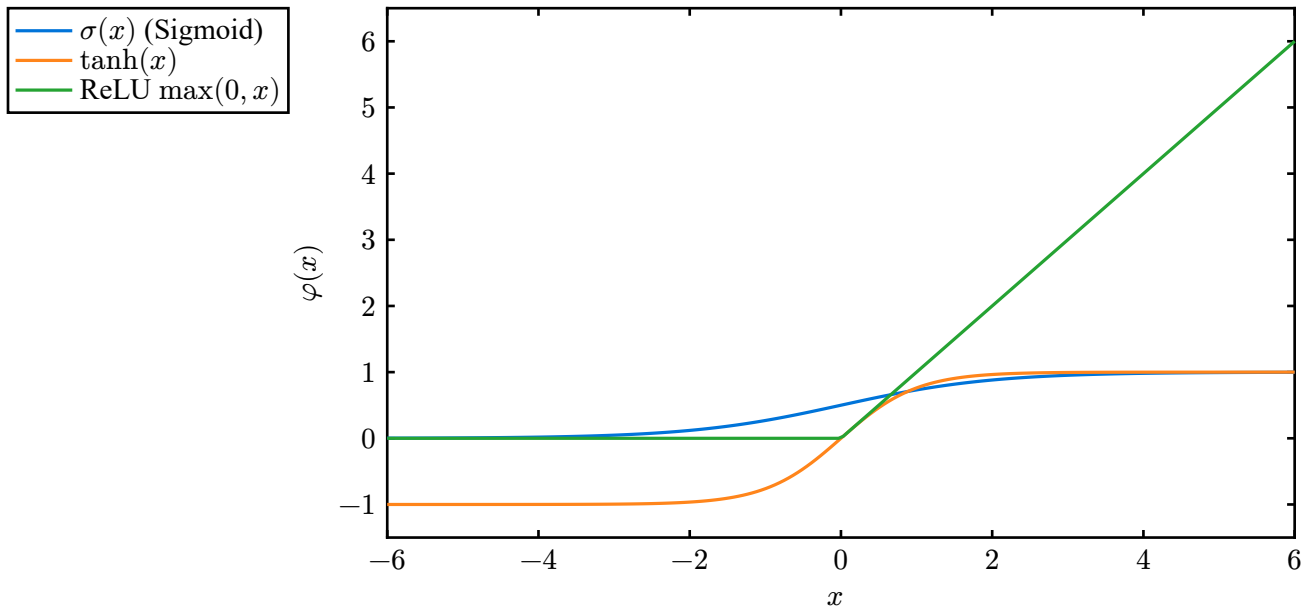


Abbildung 3: Beispielhafte Aktivierungsfunktionen (Sigmoid, tanh, ReLU).

Die Optimierung erfolgt üblicherweise iterativ. Für Gradientenabstieg gilt in kompakter Form:

$$\theta^{t+1} = \theta^t - \eta \nabla_{\theta} \mathcal{L}(\theta^t)$$

LLMs unterscheiden sich von klassischen neuronalen Netzen nicht nur durch ihre Modellgröße, sondern vor allem durch ihre Zielsetzung: Sie sind Sprachmodelle – Systeme, die Wahrscheinlichkeiten über Tokenfolgen schätzen und dadurch Texte fortsetzen, bewerten oder erzeugen können. Der Verarbeitungsablauf folgt dabei einem wiederkehrenden Muster. Zunächst wird die Texteingabe in Token zerlegt. Token sind die kleinsten Verarbeitungseinheiten des Modells. je nach Tokenisierungsverfahren können dies Wörter, Wortteile oder einzelne Zeichen sein. Der Satz „Das System prüft den Status,“ wird beispielsweise in die Folge [„Das“, „System,“, „prüft“, „den,“, „Status“] überführt. Für Quellcode gilt dasselbe Prinzip: Identifier, Schlüsselwörter und Operatoren werden in Einheiten aufgeteilt (Goodfellow et al., 2016).

Die resultierende Tokenfolge bildet den Kontext des Modells. Der Kontext umfasst alle Token, die das Modell in einem Verarbeitungsschritt gleichzeitig berücksichtigt. Aktuelle Modelle besitzen Kontextfenster von mehreren tausend bis über eine Million Token. Eingaben, die dieses Fenster überschreiten, müssen segmentiert oder komprimiert werden.

Auf Basis des Kontexts berechnet das Modell eine Wahrscheinlichkeitsverteilung über alle möglichen nächsten Token. Das gewählte Token wird an die bisherige Folge angefügt, und die erweiterte Folge dient als Eingabe für den nächsten Berechnungsschritt. Dieser Prozess wiederholt sich, bis ein Abbruchkriterium erreicht ist (z. B. ein Stopptoken oder eine maximale Ausgabelänge). Da das Modell jedes Token statistisch aus dem bisherigen Kontext ableitet, können Ausgaben sprachlich konsistent wirken, ohne dass fachliche Korrektheit gewährleistet ist. Formal lässt sich das Trainingsziel als Maximierung der bedingten Log-Likelihood beschreiben:

$$\max_{\theta} \sum_{t=1}^T \log p_{\theta}(x_t \mid x_{<t})$$

Hierbei bezeichnet $x_{<t}$ die bisherige Tokenfolge und $p_{\theta}(x_t \mid x_{<t})$ die vom Modell geschätzte Wahrscheinlichkeit für das nächste Token x_t .

In der aktuellen Modellgeneration dominiert die Transformer-Architektur (Vaswani et al., 2017), deren Kernmechanismus die Self-Attention ist. Self-Attention ermöglicht es dem Modell, bei der Verarbeitung jedes Tokens die Beziehungen zu allen anderen Token im Kontext zu gewichten. Formal wird dies als gewichtete Kombination von Value-Vektoren beschrieben, wobei die Gewichte aus Query-Key-Ähnlichkeiten berechnet werden (Vaswani et al., 2017):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

LLMs werden daher im Software Engineering eingesetzt, weil sie sowohl Artefakte in Prosa (z. B. Anforderungen, Kommentare) als auch Codeartefakte (z. B. Klassen, Funktionen, DB Schemata) verarbeiten können. Verschiedene Arbeiten versuchen daher LLM-Anwendungen nach Aufgabenklassen wie Codegenerierung, Codezusammenfassung, Fehlersuche, Testgenerierung oder Dokumentation zu ordnen (Fan et al., 2023; Hou et al., 2023; Salem et al., 2024; Zhang et al., 2023)

Aus den folgenden Eigenschaften der LLMs ergeben sich daher für diese Arbeit Konsequenzen:

- **Kontextfenster:** Modelle verarbeiten Eingaben nur bis zu einer maximalen Tokenanzahl. Längere Artefakte müssen segmentiert oder komprimiert werden.
- **Tokenisierung:** Quellcode und Fachsprache werden in Token zerlegt. Dies beeinflusst, wie gut Identifier, Struktur und Domänenterminologie repräsentiert werden.
- **Generativer Charakter:** Ausgaben sind nicht deterministisch. Temperatur, Sampling-Strategien und Promptform beeinflussen Reproduzierbarkeit.

2.2.2 LLM Training und Prompting

LLMs werden typischerweise in mehreren Phasen entwickelt. In einer Vortrainingsphase lernen Modelle aus großen Text- und Codekorpora statistische Regularitäten. Für den Einsatz als Assistenzsysteme werden Modelle häufig zusätzlich auf Anweisungen und Dialogformate ausgerichtet („instruction tuning“). Der GPT-4 Technical Report beschreibt diese Ausrichtung auf Systemebene und diskutiert Safety- und Evaluationsaspekte, ohne die vollständige Trainingspipeline offen zu legen (OpenAI et al., 2023).

Im Engineering-Kontext ist der Prompt damit nicht nur Eingabe, sondern auch ein Steuerungsinstrument. Für diese Arbeit sind vor allem folgende Hebel relevant:

Aufgabe

Ziel, gewünschtes Artefaktformat, Definition von Begriffen und Abgrenzung (z. B. „Requirement“, vs. „Designentscheidung“).

Kontextwahl

Welche Code- und Textartefakte werden bereitgestellt, und welche Teile werden bewusst ausgeblendet, um Überinterpretation zu begrenzen?

KI Leitplanken

Belegpflicht, Kennzeichnung unsicherer Aussagen, feste Templates, DOs and DONTs.

Da LLMs ein begrenztes Kontextfenster besitzen, wird in Forschung und Praxis häufig Retrieval-Augmented Generation (RAG) eingesetzt: Relevante Textstellen werden zunächst über Suche/Retrieval ausgewählt und anschließend als Kontext in die Generierung eingebracht. Lewis et al. (2020) beschreiben dieses Grundprinzip für wissensintensive Aufgaben. Für Requirements-Extraktion aus Legacy-Code ist RAG naheliegend, weil relevante Regeln, Konfigurationen und UI-Strings über große Repositories verteilt sind und eine „Alles in den Prompt“-Strategie nicht skaliert.

Prompting-Strategien wie Chain-of-Thought können die Qualität komplexer Ableitungen verbessern, bergen im Requirements-Kontext jedoch ein Risiko: Längere Begründungen können plausibel wirken und dadurch Fehlannahmen stabilisieren. Wei et al. (2022) zeigen Chain-of-Thought als wirksame Prompttechnik.

Für diese Arbeit folgt daraus vor allem, dass Begründungen stets mit Artefaktbelegen gekoppelt werden müssen und nicht als eigenständige Evidenz gelten.

2.2.3 Qualitätsrisiko Halluzinationen

Halluzinationen bezeichnen Ausgaben, die syntaktisch korrekt und plausibel wirken, aber nicht durch Eingabedaten oder Weltwissen gedeckt sind. Ji et al. (2023) liefern die Begrifflichkeit und diskutieren Detektions- und Lösungsansätze. Für Requirements ist die Gefahr besonders kritisch, weil falsche Requirements nicht als „Bug“, auffallen, sondern als scheinbar saubere Spezifikation in Architektur- und Funktionsentscheidungen einfließen können.

Zusätzlich zu Halluzinationen sind zwei weitere Verlässlichkeitsthemen relevant:

Daten- und Domänenbias

Modelle spiegeln Verteilungen und Annahmen aus Trainingsdaten wider (Bender et al., 2021). Taucht eine falsche Aussage in Trainingsdaten häufig auf, wird sie vom Modell übernommen und als Wahrheit ausgegeben.

Reproduzierbarkeit

Kleine Promptänderungen oder Parameterunterschiede können zu unterschiedlichen Ergebnissen führen. Für einen engineeringfähigen Prozess sind daher Leitplanken (z. B. feste Templates, deterministische Einstellungen, versionierte Prompts) notwendig.

Für diese Arbeit folgt daraus, dass LLM-Ausgaben im Requirements-Kontext nicht als Wahrheit,, sondern als Vorschlag zu behandeln sind. Erst durch Traceability (Belege) und Validierung (Expertenreview, Laufzeitchecks) wird aus einer Hypothese eine belastbare Anforderung.

2.2.4 LLMs im Requirements Engineering: Stand der Forschung

Aktuelle Arbeiten untersuchen, wie LLMs Texte erzeugen, umformulieren und strukturieren können, und verschieben den Schwerpunkt damit gegenüber klassischen NLP/IR-Ansätzen mit begrenzten Ausgaben (Labels, Links, Hinweise) von reinen Analysen der Zusammenhänge hin zu tatsächlicher Formulierung von Requirements. Dieser Übergang ist aber nicht ohne Risiken: Einerseits entsteht ein großes Potential, andererseits steigt das Risiko, dass sprachlich „gute“ Texte als Spezifikation akzeptiert werden, obwohl die fachliche Basis unzureichend oder fehlerhaft ist (siehe Haluzinationen) (Hemmat et al., 2025; Ji et al., 2023).

Eine systematische Übersicht ordnet die LLM-Nutzung im RE dabei entlang klassischer Prozessschritte ein und nennt als wiederkehrende Problemfelder *Qualitätssicherung, Nachvollziehbarkeit und Domänenabhängigkeit* (Hemmat et al., 2025). Ein weiteres Review zum LLM-Einsatz im Requirements Engineering diskutiert den Einsatz entlang typischer RE-Aktivitäten (Analyse, Spezifikation, Validierung) und hebt als zentrale Herausforderungen inkonsistente Ergebnisse durch limitierte Kontextgrößen und begrenztes Domänenwissen hervor (Marques et al., 2024).

Dabei lassen sich aktuelle LLM-Arbeiten grob folgenden Themen zusammenfassen:

Strukturierung und (Re-)Formulierung von Requirements

Untersucht wird, wie LLMs natürlichsprachliche Anforderungen in strukturiertere Formen überführen können (Norheim & Rebentisch, 2024), sowie die automatische Umstrukturierung von Software Requirements Specifications mit dem Ziel, Standardkonformität zu erhöhen (Okamoto & Kusumoto, 2025).

Qualitätsunterstützung und Analyse

ChatGPT wurde für die Inkonsistenzdetektion in naturalsprachlichen Requirements evaluiert (Fantechi et al., 2023); weitere Arbeiten untersuchen LLM-gestützte Assistenz zur Verbesserung der Requirements-Vollständigkeit (Luitel et al., 2024).

Anforderungserhebung (Elicitation) und Perspektivenwechsel

LLMs können zur Generierung wertorientierter User Stories als „Inspirationsimpulse“ eingesetzt werden (Marczak-Czajka & Cleland-Huang, 2023). Diese Richtung ist für Reverse Requirements Engineering insofern relevant, weil sie zeigt, wie LLMs fehlende Stakeholder-Sichten ergänzen können, ohne den Code als Primärbeleg zu ersetzen.

Domänenspezifische Requirements (Safety/Compliance)

Betrachtet wurden LLMs bei der Engineering-Unterstützung von Safety Requirements im Kontext autonomen Fahrens (Nouri et al., 2024) sowie für rechtliche Compliance- und Regulationsanalyse (Hassani, 2024). Solche Arbeiten verdeutlichen, dass LLMs nicht nur Text umformulieren, sondern auch regulatorische Anforderungen (Normen, Regeln) einbinden können.

Insgesamt ist die Studienlage bisher uneinheitlich. Viele Arbeiten sind kurze Workshopbeiträge oder erste Vorstudien mit kleinen Datensätzen, die automatische Messungen mit Experteneinschätzungen mischen. Auch die verwendeten Prompts, Modellversionen und Einstellungen sind selten einheitlich dokumentiert, wodurch sich Ergebnisse schwer wiederholen lassen (Fan et al., 2023; Hemmat et al., 2025).

Für Reverse Requirements Engineering lässt sich der Nutzen damit präzisieren: LLMs können Kandidaten-Requirements aus großen Artefaktmengen (Code, Kommentare, UI-Strings, Konfiguration) herauslesen und verdichten und in eine konsistente Spezifikationsform überführen. Die fachliche Belast-

barkeit entsteht jedoch erst durch Traceability zu Codebelegen und die Validierung durch Experten, insbesondere bei Safety-, Compliance- und Abrechnungslogik.

2.2.5 Absicherung: Human-in-the-loop, Belege und Prozesskontrollen

Die Literatur legt nahe, dass LLMs im Software Engineering dann robust eingesetzt werden können, wenn sie in einen Prozess eingebettet sind, der Fehler systematisch begrenzt (Fan et al., 2023; Hemmat et al., 2025). Für die Requirements-Extraktion aus Legacy-Code sind folgende Kontrollen praxisnah:

Belegpflicht (Evidence-First)

Jedes generierte Requirement erhält mindestens einen konkreten Beleg (Datei/Komponente/Query/GUI-String) sowie eine kurze Begründung, warum der Beleg die Aussage trägt.

Trennung von Fakt und Interpretation

Technische Fakten (z. B. „Status = ‚Closed‘ verhindert Update,“) werden getrennt von fachlicher Interpretation (z. B. „Abgeschlossene Aufträge sind schreibgeschützt“) dokumentiert.

Mehrstufige Validierung

Automatische Checks (z. B. Linting auf Verbformen, Konsistenzregeln) werden mit Expertenreview kombiniert.

Reproduzierbarkeit

Versionierung von Promptvorlagen, Modellversionen und Kontextzuschnitten, um Ergebnisse vergleichbar zu machen.

Diese Kontrollen adressieren nicht alle Risiken, reduzieren aber die typischen Fehlerklassen (Halluzination, Überinterpretation, fehlende Konsistenz) und schaffen die Grundlage für eine belastbare Evaluation.

2.2.6 Qualitätsbewertung und Messgrößen im Requirements-Kontext

Die Qualität von LLM-Ergebnissen wird in vielen Arbeiten mit allgemeinen Textmetriken oder aufgabenspezifischen Benchmarks bewertet. Für die Requirements-Extraktion aus Code reichen solche Metriken nicht aus, da es hier weniger um sprachliche Ähnlichkeit geht, sondern um fachliche Korrektheit, Prüfbarkeit und Nachvollziehbarkeit (Hemmat et al., 2025; Marques et al., 2024). Eine sinnvolle Bewertung orientiert sich daher an RE-Kriterien und unterscheidet drei Dimensionen:

Statement-Qualität

Ist ein Requirement eindeutig, vollständig im Satzbau, frei von nicht belegten Annahmen und mit Akzeptanzkriterium bzw. Prüfidée versehen?

Set-Qualität (Vollständigkeit)

Deckt die Menge der Requirements alle relevanten Prozesse und Varianten vollständig ab, ist sie in sich konsistent und frei von Doubletten? Die Vollständigkeit ist dabei die zentrale Eigenschaft, weil ein fehlendes Requirement im Migrationskontext zu Funktionsverlust führen kann, während Inkonsistenzen oder Doubletten in einem Review nachträglich aufgelöst werden können.

Traceability-Qualität

Sind Belege reproduzierbar auffindbar (z. B. Dateipfad, Methode, SQL-Query), und lässt sich die Ableitung von „Beleg → Requirement,“ nachvollziehen?

Für Legacy-Migrationen ist zudem die Fehlerkostenperspektive entscheidend. Ein fehlendes Requirement kann zu Funktionsverlust führen, ein falsches Requirement kann zu fehlerhaften Designentscheidungen führen, und ein unpräzises Requirement verursacht Review- und Nacharbeit. Daraus folgt eine pragmatische Bewertung: Requirements mit hoher Migrationskritikalität (z. B. Sicherheitsregeln, Abrechnungslogik, Berechtigungen) sollten strengere Evidenzanforderungen und intensivere Reviews erhalten als periphere Funktionen. Dieses Prinzip ist kompatibel mit der risikobasierten Priorisierung von Qualitätsanforderungen (Glinz, 2008) und lässt sich auf Funktionsanforderungen übertragen.

2.3 Legacy-Modernisierung und Stand der Forschung

2.3.1 Charakteristika von Legacy-Systemen

Legacy-Systeme sind nicht allein durch ihr Alter, sondern durch ihren Kontext definiert. Sie tragen geschäftskritische Funktionen, sind über lange Zeit erweitert worden und weisen oft starke technische und organisatorische Abhängigkeiten auf. Typische Merkmale sind enge Kopplung, heterogene und zersplitterte Technologien, schwer austauschbare oder sogar schon abgekündigte Komponenten und unzureichende Dokumentation (Bisbal et al., 1999). Gerade die fehlende Dokumentation ist für Modernisierungsvorhaben problematisch, weil Entscheidungen ohne belastbare Anforderungsbasis zu Funktionsverlusten und Fehlentscheidungen führen können.

Im ERP-Kontext verschärfen sich diese Merkmale häufig durch eine ausgeprägte Domänenkomplexität, da Geschäftsregeln variantenreich und teilweise kundenspezifisch sind. Hinzu kommt eine starke Daten-zentrierung. Prozesse hängen stark von Datenmodellen, Stammdatenqualität und historisch gewachsenen Datenkonventionen ab. Verstärkt wird dies durch eine hohe Integrationsvielfalt, da Schnittstellen zu Drittsystemen (z. B. Buchhaltung, Shop, Dokumentenmanagement) über Jahre organisch entstanden sind.

Damit wird nachvollziehbar, warum Requirements-Extraktion aus der Codebasis nicht nur ein Dokumentationsprojekt ist. Es dient zeitlich auch zur Risikoreduktion.

2.3.2 Modernisierungsstrategien

Eine Modernisierung kann durch unterschiedliche Methoden umgesetzt werden, von „Lift-and-Shift“ bis zur vollständigen Neuimplementierung „auf der grünen Wiese“. Die Literatur betont wiederholt, dass die Wahl einer Strategie von Risiko, Zielarchitektur und verfügbaren Ressourcen abhängt und daher explizit geplant werden sollte (Sneed, 1995). Eine zentrale Aussage ist dabei, dass Reengineering nicht als rein technischer Umbau verstanden werden darf. Ohne fachliche Leitplanken entstehen technische Verbesserungen, die am Bedarf vorbeilaufen oder bestehende Fachlogik unabsichtlich verändern. Zudem können gewonnene fachliche Erkenntnisse ohne klare Dokumentation und Nachvollziehbarkeit nicht in die Zielarchitektur überführt werden, was zu einem erneuten Anforderungsdefizit führt.

Aus Sicht dieser Arbeit lassen sich Modernisierungsmethodiken pragmatisch entlang zweier Dimensionen einordnen: (1) Wie stark wird die bestehende Implementierung weitergenutzt? (2) Wie stark wird die Zielarchitektur verändert? Daraus ergeben sich typische Strategietypen, die in der Praxis auch kombiniert auftreten:

- a) **Weiterbetrieb mit Hülle (Wrapping):** Die Legacy-Logik bleibt bestehen, wird aber über neue Schnittstellen oder eine neue GUI zugänglich gemacht. Vorteil ist geringe Eingriffstiefe; Nachteil ist, dass technische Schulden und Engpässe erhalten bleiben.
- b) **Schrittweise Modularisierung:** Teile der Legacy-Anwendung werden sukzessive in neue Komponenten überführt, während andere Teile weiterlaufen. Vorteil ist Risikostreuung und frühe Nutzenrealisierung; Nachteil ist erhöhte Integrationskomplexität während der Übergangsphase.
- c) **Reengineering/Refactoring:** Die bestehende Logik wird strukturell überarbeitet (z. B. Entkopplung, Schichten, bessere Testbarkeit), ohne den Funktionsumfang grundsätzlich zu verändern. Vorteil ist bessere Wartbarkeit; Nachteil ist hoher Analyseaufwand, gerade ohne Requirementsbasis.
- d) **Neuimplementierung mit Funktionsparität:** Die Legacy-Logik wird auf neuer Technologie nachgebaut, häufig mit dem Anspruch, zunächst funktional äquivalent zu sein. Vorteil ist saubere Zielarchitektur; Nachteil ist die hohe Abhängigkeit von vollständigen, korrekten Requirements.

Für ein ERP-System ist die Wahl einer Strategie stark datengetrieben. Datenmodelle und Businesslogik definieren die Leitplanken einer Migration. Damit steigt die Wichtigkeit von Requirements, die Datenobjekte, Zustandsmodelle und Integrationspunkte ausdrücken. Besonders kritisch sind dabei

Anforderungen, die in der Legacy-Implementierung als implizite Konvention existieren (z. B. Statuscodes, historische Sonderfälle, kundenspezifische Maskenlogik), weil sie ohne gezielte Extraktion und Validierung leicht verloren gehen.

2.3.3 Zielarchitekturen: Web, Cloud und „Cloud-native,,

Die Modernisierung vieler Legacy-Anwendungen zielt auf webbasierte, plattformunabhängige Oberflächen und auf Betriebsmodelle, die Skalierung, automatisiertes Deployment und schnelle Iteration unterstützen. Eine systematische Mapping Study fasst zusammen, welche Merkmale cloud-nativer Anwendungen in Forschung und Praxis wiederkehren (Kratzke & Quint, 2017). Dazu zählen typischerweise automatisierte Bereitstellung, resiliente Komponenten, horizontale Skalierung und eine stärkere Trennung von Build- und Run-Umgebungen.

Im selben Zusammenhang werden Microservices häufig als Architekturstil diskutiert. Eine Analyse der Forschung zu Microservices zeigt wiederkehrende Problemfelder, unter anderem die Wahl der richtigen Servicegranularität, die erhöhte Komplexität im Betrieb und Anforderungen an Observability (Pahl & Jamshidi, 2016). Für Migrationsprojekte ist daraus eine pragmatische Schlussfolgerung ableitbar: Modularisierung ist ein Ziel, erzeugt aber zugleich neue Anforderungen (z. B. Deployment-Pipelines, Monitoring, Sicherheitskonzepte), die im Requirements-Set sichtbar sein müssen.

Für die Requirementsentwicklung bedeutet diese neue Zielarchitektur eine Verschiebung des Schwerpunktes. Während in klassischen Server/Client-Architekturen die fachliche Funktionslogik oft dominiert, rücken in Web- und Cloud-Kontexten auf den Betrieb bezogene (Deployment, Monitoring, Skalierung) und sicherheitsrelevante Qualitätsmerkmale (SaaS, Multi-Tenancing) stärker in den Vordergrund. ISO/IEC 25010:2011 bietet hierfür eine hilfreiche Taxonomie (ISO/IEC, 2011). Für Modernisierungsvorhaben lassen sich vor allem folgende Qualitätsmerkmale als wiederkehrend beobachten:

Sicherheit

Identitäten, Rollenmodelle, Mandantenfähigkeit, Auditierbarkeit.

Zuverlässigkeit

Fehlerresistenz, Wiederanlauf, Degradationsverhalten.

Performance-Effizienz

Antwortzeiten, Lastverhalten, Skalierungsgrenzen.

Wartbarkeit

Änderbarkeit, Testbarkeit, Modularität und technische Schuld.

Kompatibilität und Interoperabilität

Schnittstellenstabilität, Integrationsfähigkeit mit Drittsystemen.

Diese Merkmale sind nicht neu, ihre Sichtbarkeit im Projekt nimmt jedoch zu, weil Cloud- und Webbetrieb ein engeres Zusammenspiel von Entwicklung und Betrieb erzwingt.

Für das Reverse Requirements Engineering im Rahmen dieser Arbeit folgt daraus, dass der Blick auf die Legacy-Codebasis systematisch um Betriebs- und Sicherheitsanforderungen ergänzt werden muss, auch wenn diese im Code nur indirekt sichtbar sind (z. B. über Deployment-Skripte, Konfigurationen, Logging-Policies oder Rechteprüfungen).

2.3.4 Stand der Forschung: KI-Unterstützung in Modernisierungsvorhaben

Die Forschung zu LLM-Unterstützung bei Modernisierungen steht im Vergleich zu klassischen Reengineering-Ansätzen noch an den Anfängen. Die Übersichten zu LLM4SE (Fan et al., 2023; Hou et al., 2023) zeigen, dass ein Teil der Arbeiten auf Codeverständnis, Dokumentation und Artefakttransformation zielt. Spezifisch für Requirements Engineering zeigen Reviews und SLRs erste konkrete Forschungsrichtungen (Hemmat et al., 2025; Marques et al., 2024).

Aus dieser Literatur lassen sich zwei Aussagen ableiten. Zum einen sind LLMs besonders stark in der Strukturierung und sprachlichen Formulierung. Also dort, wo aus heterogenen Artefakten ein konsistenter Text entstehen muss. Zum anderen benötigen LLMs technische und organisatorische Sicherungen, wenn Ergebnisse als Entscheidungsgrundlage in Migrationen dienen sollen (z. B. Belege, Review, reproduzierbarer Prozess).

Dies ist auch die Zentrale Motivation dieser Arbeit: Eine Legacy-Modernisierung benötigt belastbare Requirements, die im Legacy-Kontext oft fehlen. LLMs sind als Assistenz zur Rekonstruktion naheliegend, müssen jedoch methodisch so eingesetzt werden, dass Verlässlichkeit und Nachvollziehbarkeit systematisch erhöht werden.

2.3.5 Kapitelzusammenfassung und Anschluss

Die drei Themenblöcke dieses Kapitels greifen ineinander. Requirements Engineering liefert Kriterien, um Anforderungen prüfbar und nachvollziehbar zu formulieren (ISO/IEC/IEEE, 2018). Reverse Requirements Engineering überträgt diese Kriterien in einen Kontext, in dem Anforderungen aus bestehenden Artefakten rekonstruiert werden müssen. Large Language Models können bei dieser Rekonstruktion unterstützen, sind aber fehleranfällig und benötigen Prozesskontrollen, vor allem gegen Halluzinationen und Überinterpretation. Legacy-Modernisierung schließlich liefert die praktische Motivation und zeigt, warum eine belastbare Anforderungsbasis migrationskritisch ist (Bisbal et al., 1999; Sneed, 1995).

3 Fallstudie c-entron GmbH

Dieses Kapitel beschreibt die Anwendung auf die sich diese Arbeit bezieht. Betrachtet wird die Windows-basierte ERP-Software der c-entron GmbH, die auf eine webbasierte Plattform überführt werden soll. Ziel ist es, die fachlichen und technischen Rahmenbedingungen so darzustellen, dass die Anforderungen an ein KI-gestütztes Reverse Requirements Engineering nachvollziehbar werden. Zunächst werden Unternehmenskontext und Legacy-Software eingeordnet, anschließend folgen Migrationsstrategie und spezifische Herausforderungen.

3.1 Unternehmenskontext und Legacy-Software

3.1.1 Unternehmens- und Domänenkontext

Die c-entron GmbH (Ulm) entwickelt und betreibt eine ERP-Suite für IT-Systemhäuser. Die Software ist über mehrere Jahrzehnte gewachsen, der Funktionsumfang entsprechend breit. Mit der historischen Tiefe gehen produktionsnahe Sonderfälle und kundenbezogene Varianten einher, die im Tagesgeschäft funktionieren, aber nur teilweise dokumentiert sind.

Für diese Arbeit ist die Software relevant, weil sie Anforderungen in einer Form enthält, die für Reverse Requirements Engineering typisch schwierig ist. Geschäftskritische Prozesse wie Auftragsabwicklung, Lager und Fakturierung sind nicht nur in einzelnen Masken abgebildet, sondern in Validierungen, Statusübergängen und Datenmodellrestriktionen verteilt. Hinzu kommt die Kopplung an weitere Anwendungen über Import- und Exportschnittstellen. Aus diesen Artefakten müssen für die Modernisierung stabile Geschäftsregeln und Datenbeziehungen abgeleitet werden, die als Grundlage einer webbasierten Neuimplementierung dienen.

3.1.2 Technologischer Ist-Stand

Die ERP-Suite ist als Windows-Anwendung in klassischer Client/Server-Architektur über Jahrzehnte gewachsen. Für die Modernisierung sind nicht das Alter, sondern die Kopplung der Komponenten und die lückenhafte Dokumentation entscheidend. Folgende Merkmale sind für die Fallstudie besonders relevant:

Enge Kopplung zwischen UI, Fachlogik und Persistenz

Geschäftsregeln sind nicht in einer Schicht isoliert, sondern verteilen sich über mehrere Komponenten. So liegen Validierungen im Frontend-Code, während weitere Regeln als Trigger in der MSSQL-Datenbank realisiert sind.

Implizite Regeln in Code und Daten

Ein Teil der Anforderungen ist nicht explizit dokumentiert, sondern ergibt sich aus Validierungen, Standardwerten und Datenmodellannahmen.

Evolution über lange Zeiträume

Funktionserweiterungen und Fehlerkorrekturen haben zu Sonderfällen und Workarounds geführt, die fachlich begründet, aber selten als Requirement festgehalten sind. Regeln zum gleichen Fachthema (z. B. Preisberechnung) werden auf unterschiedlichen Masken nicht einheitlich angewendet, was die genaue Definition der Regel erschwert.

Heterogene Artefakte

Neben Quellcode existieren Konfigurationen, UI-Texte, Reportdefinitionen sowie Import- und Exportlogiken, die Anforderungen indirekt spiegeln.

Die Software ist damit ein geeigneter Gegenstand für Reverse Requirements Engineering. Die Anforderungen liegen nicht als gesammelte Dokumentation vor, sondern müssen erst rekonstruiert werden.

3.1.3 Dokumentations- und Wissenslage

Für das Modernisierungsvorhaben liegt keine schriftliche Anforderungsdokumentation vor. Vollständigkeit, Verifizierbarkeit und Traceability als zentrale Qualitätskriterien des Requirements Engineering müssen daher aus anderen Quellen hergestellt werden. Fachliche Regeln sind primär in der **Implementierung und im Laufzeitverhalten** gebunden. Änderungsanlässe, Bugfixes und Kundenanpassungen lassen sich ergänzend aus der **Change-Historie** in Issue-Tracker, Commits und Releases rekonstruieren. Ein wesentlicher Teil liegt zudem als **Erfahrungswissen einzelner Mitarbeiter** vor; dieses Domänenwissen ist personenbezogen und stellt ein Risiko bei Personalwechsel dar.

Für diese Arbeit folgt daraus, dass der Wert eines KI-gestützten Reverse Requirements Engineering nicht in der Generierung „gut klingender“, Spezifikationstexte liegt, sondern in der systematischen Extraktion belegbarer Aussagen mit Referenz zu existierenden Artefakten, die als Requirement prüfbar sind und die spätere Migration absichern.

3.1.4 Relevante Artefakte der Fallstudie

Für das Vorgehen in den folgenden Kapiteln ist es hilfreich, die vorhandenen Artefakte zu ordnen. In der ERP-Modernisierung sind insbesondere folgende Artefaktklassen Träger von Requirements:

- a) **Quellcode:** Implementierte Regeln, Berechtigungen, Statuslogik, Datenzugriffe.
- b) **Konfiguration und Parameter:** Feature-Schalter, Mandantenparameter, systemweite Defaults.
- c) **UI- und Reportartefakte:** Feldbezeichnungen, Validierungstexte, Druck- und Exportformate.
- d) **Datenstrukturbezogene Artefakte:** Datenmodelle, Constraints, Referenzen, historisierte Strukturen.
- e) **Projektartefakte:** Tickets, Release Notes, Testfälle, Migrationsnotizen.

Quellcode und Datenstrukturen liefern den belastbarsten Beleg für eine Regel, während Tickets und Release Notes vor allem den Anlass einer Änderung dokumentieren. Die fachliche Validierung wird gezielt auf risikoreiche oder unsichere Aussagen gelenkt.

3.2 Migrationsstrategie und spezifische Herausforderungen

3.2.1 Ziel der Modernisierung

Ziel der Modernisierung ist eine webbasierte SaaS-Plattform, die sowohl Cloud- als auch On-Premise-Betrieb unterstützt. Neben der Funktionsäquivalenz zur bestehenden ERP-Suite stehen nicht-funktionale Anforderungen wie Skalierbarkeit, Mandantenfähigkeit und KI-Anbindung im Vordergrund. Konkret leiten sich für das Fallunternehmen die folgenden Anforderungen ab:

Cloud- und On-Premise-Fähigkeit

Die Anwendung wird containerisiert ausgeliefert und ist damit unabhängig von einem bestimmten Infrastrukturanbieter. Neben dem zentralen SaaS-Betrieb bleibt eine Einzelinstallation beim Kunden möglich.

Multi-Tenant-Fähigkeit

Mehrere Mandanten teilen sich eine gemeinsame Datenbank, sodass auch kleinere Kunden ohne vollen Installationsaufwand bedient werden können. Für größere Kunden ist weiterhin eine Einzelinstallation vorgesehen.

Datenbanksystem-Unabhängigkeit

Geschäftslogik und Abfragen werden möglichst vollständig im Code abgebildet und nicht im DBMS. Damit sinken Migrationsaufwände, und im Multi-Tenant-Betrieb lassen sich günstige oder kostenlose Datenbanksysteme einsetzen.

Skalierbarkeit

Das Backend ist über Microservices und Load-Balancing horizontal skalierbar, um auch bei größeren Kunden performant zu bleiben.

KI-Fähigkeit

Schnittstellen zu KI-Systemen (z. B. MCP-Server, RAG-Embeddings) sind von Beginn an in der Architektur vorgesehen.

Web-Frontend

Eine responsive Web-Oberfläche ermöglicht die geräteherstellerunabhängige Nutzung.

Zentrale Datenverwaltung

Das Basissystem stellt eine eigenständige Oberfläche zur Benutzer- und Kundenverwaltung bereit und dient auch ohne ERP-Funktionalität als Basis für weitere Produkte der Firma.

Als Performance-Ziel wird ein Bereich von 5 bis 100 gleichzeitigen Nutzern pro Mandant festgelegt. Kleinere Installationen mit 1 bis 5 Nutzern sowie größere Installationen oberhalb von 100 Nutzern werden ebenfalls unterstützt, bleiben aber opportunistische Ziele.

3.2.2 Strategische Optionen und Abgrenzung

Migrationsstrategien für Legacy-Software reichen von leichtgewichtigen Ansätzen wie Refactoring bis zur schrittweisen Neuimplementierung zentraler Funktionen. Gegenstand dieser Arbeit ist nicht die vollständige Migrationsplanung, sondern die Frage, wie eine belastbare Requirementsbasis für die Umsetzung erzeugt wird.

Schwerpunkt ist die **Rückgewinnung und Strukturierung** von Requirements aus bestehenden Artefakten. Architektur- und Implementierungsentscheidungen der Zielplattform werden nur insoweit diskutiert, als sie Requirements beeinflussen, etwa bei Sicherheits- und Betriebsanforderungen. Die technische Migration selbst, also Datenmigration, Refactoring im Detail, Reimplementierung oder Release-Planung, wird lediglich als Randbedingung mit betrachtet.

3.2.3 Spezifische Herausforderungen im Fallunternehmen

Aus dem Ist-Zustand ergeben sich mehrere Herausforderungen, die das Vorgehen des Reverse Requirements Engineering unmittelbar prägen:

Randfäll und Varianten

Ein erheblicher Teil des Systemumfangs steckt in implementierten Sonderfällen und Varianten. Diese sind im Code zwar nachvollziehbar, aber selten dokumentiert und zur Laufzeit auch jeweils nur einzeln zu analysieren da sie sich oft gegenseitig ausschließen.

Daten und Logische Redundanz

Historisch wurden zusätzliche Funktionen oft inimplementiert indem Code oder Masken hinzugefügt wurden ohne alten code zu verändern. Daher kommt es vor, dass die gleiche Anforderung auf Unterschiedlichen Masken oder Berechnungspfaden mehrfach unterschiedliche implementiert wurde.

Implizite Geschäftsregeln

Geschäftslogik ist häufig als Prüf- und Statuslogik oder über Randbedingungen in Eingabefeldern realisiert. Eine Zuordnung zu anderen zusammengehörigen Regeln fällt hier schwer.

Kontextbasierte Logik

Funktionalitäten und Logik sind oft nur lose über historische Zustände verbunden. Eine Regel kann daher nur bei Betrachtung eines vollständigen historischen Datensatzes im Kontext abgeleitet werden.

Nicht-funktionale Anforderungen

Mit dem Wechsel auf einen neuen Tech-Stack ist zu prüfen, welche der bisherigen nicht-funktionalen Anforderungen weiterhin Gültigkeit haben und welche neu definiert werden müssen. Annahmen zu Performance, Sicherheit oder Verfügbarkeit gelten unter veränderten Architektur- und Betriebsbedingungen nicht automatisch weiter.

3.2.4 Konsequenzen für das Vorgehen in dieser Arbeit

Aus den Herausforderungen ergeben sich vier konkrete Anforderungen an das Vorgehen. Zunächst gilt eine **Belegpflicht**. Jede extrahierte Anforderung muss auf ein Artefakt wie eine Datei, ein Modul, ein Datenobjekt oder einen UI-Text zurückgeführt werden können. Aussagen, die nicht eindeutig aus Artefakten ableitbar sind, werden **explizit als Hypothesen markiert** und priorisiert validiert. Da Artefakte verteilt sind, ist eine **Segmentierung und Kontextsteuerung** notwendig, um Überinterpretation zu reduzieren. Schließlich ist eine fachliche Validierung durch einen **Human-in-the-loop** zwingend, da „plausible“, Textausgaben kein hinreichender Beweis für fachliche Korrektheit sind.

4 Konzeption und methodisches Vorgehen

Dieses Kapitel beschreibt die Methodik, mit der die in Kapitel 1 beschriebenen Ziele und Forschungsleitfragen beantwortet werden sollen. Ausgangspunkt ist das methodische Design. Aus diesem Design leiten sich alle weiteren methodischen Entscheidungen ab. Vorausgegangene Proof-of-Concept-Läufe haben einzelne Aspekte des Vorgehens informell erprobt und das hier dargestellte Vorgehen geprägt, sie sind aber nicht Gegenstand der Auswertung. Die eigentliche Untersuchung wird in den folgenden Abschnitten geplant und in den folgenden Kapiteln durchgeführt und bewertet.

4.1 Werkzeug-Grundlagen: Agentische CLIs, Agenten, MCP und lokale Inferenz

Bevor der eigentliche Versuchsaufbau beschrieben wird, werden die zentralen Werkzeug-Begriffe geklärt, an denen die spätere Versuchsreihe variiert. Sie sind nicht nur Hilfsmittel der Durchführung, sondern definieren genau die Stellschrauben, die in den Versuchen V1 bis V3 systematisch hinzugenommen werden. Die folgende Darstellung beschränkt sich auf das für diese Arbeit relevante Maß und ersetzt keine vollständige Werkzeugdokumentation. Das Zusammenspiel der vier Bausteine ist in Abbildung 4 skizziert.

Agentische CLIs

Eine LLM CLI ist ein Kommandozeilen-Werkzeug, das einem LLM den direkten Zugriff auf die lokale Arbeitsumgebung erlaubt. Es kann nicht nur online Artefakte erzeugen, sondern auch lokale Dateien lesen und bearbeiten sowie Shell-Befehle ausführen oder lokal installierte Tools aufrufen. Innerhalb einer Ausführungsschleife entscheidet das LLM selbst, welches Werkzeug es auf dem lokalen Computer aufrufen möchte. Die CLI stellt dafür lediglich die Laufzeitumgebung und den Zugriff auf die lokalen Werkzeuge bereit. In dieser Arbeit kommen Claude Code, die Codex-CLI sowie die Qwen Code CLI zum Einsatz.

Agenten und Agentendateien

LLM Agenten sind rollenspezialisierte Konfigurationen (Prompts), die innerhalb einer agentischen CLI als Sub-Prozesse aufgerufen werden. Sie werden typischerweise als Markdown-Dateien abgelegt und enthalten drei Bestandteile: einen Auftrag (was tut der Agent), eine eigene Systemanweisung (wie tut er es) und ein begrenztes Toolset (womit darf er es tun). Die aufrufende CLI startet einen solchen Agenten mit eigenem Kontextfenster, sodass dessen Zwischenergebnisse den Hauptkontext nicht überfluten.

Model Context Protocol (MCP)

Das Model Context Protocol ist ein 2024 von Anthropic veröffentlichter offener Standard zur Anbindung externer Werkzeuge und Datenquellen an LLM-Clients (Anthropic, 2026). Ein MCP-Server kapselt konkrete Fähigkeiten, etwa Symbol-Navigation im Quellcode, Datenbankzugriff oder GUI-Manipulation, hinter einer einheitlichen Schnittstelle. Ein MCP-Client wie Claude Code entdeckt die angebotenen Werkzeuge zur Laufzeit und ruft sie kontrolliert auf, um ein deterministisches Ergebnis zu erhalten, anstatt den Aufruf direkt durch das LLM zu generieren. So stellt etwa ein MCP-Server für lesenden Datenbankzugriff eine fest definierte SQL-Schnittstelle bereit, über die das LLM Tabellenstrukturen und Datensätze reproduzierbar abrufen, anstatt sie aus dem allgemeinen Modellwissen zu rekonstruieren.

Lokale Inferenz-Runtimes

Eine lokale Inferenz-Runtime ist eine Software, die LLMs auf eigener Hardware ausführt und ihre Funktion über einen OpenAI-kompatiblen HTTP-Endpunkt bereitstellt. In dieser Arbeit kommt LM Studio zum Einsatz, das Modelle wie Qwen 3.5 Coder lokal ausführen kann.

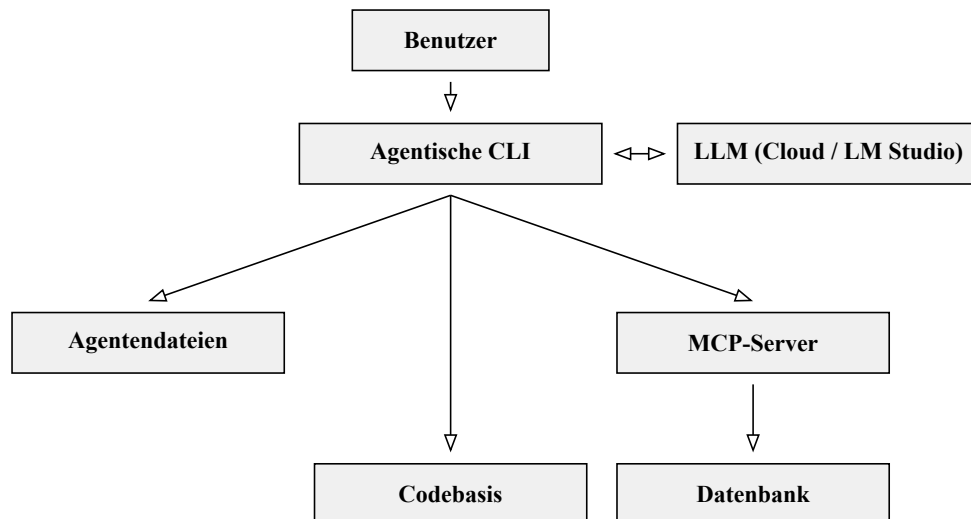


Abbildung 4: Hierarchie der Werkzeuge. Der Anwender ruft die agentische CLI auf. Diese übergibt das initiale Prompt und das verfügbare Toolset an das LLM und erhält dessen Antworten samt Tool-Aufrufen zurück. Die CLI führt die vom LLM erbetenen Aufrufe aus. Die Codebasis ist direkt über die CLI zugänglich. Der Zugriff auf die Datenbank erfolgt ausschließlich über einen MCP-Server.

4.2 Methodisches Design im Überblick

Das Vorgehen ist entlang der vier Forschungsleitfragen aus Kapitel 1 strukturiert. Diese werden im Folgenden mit Frage 1 (Steuerung und Reproduzierbarkeit), Frage 2 (KI-Extraktion und Stakeholder-Input), Frage 3 (Qualitätsbewertung) und Frage 4 (Chancen, Grenzen und Risiken) bezeichnet. Aus jeder Leitfrage folgt unmittelbar eine Datenquelle und ein Auswertungsweg. Damit ist sichergestellt, dass die methodischen Bausteine nicht nachträglich auf die Fragen abgebildet werden, sondern aus ihnen hervorgehen.

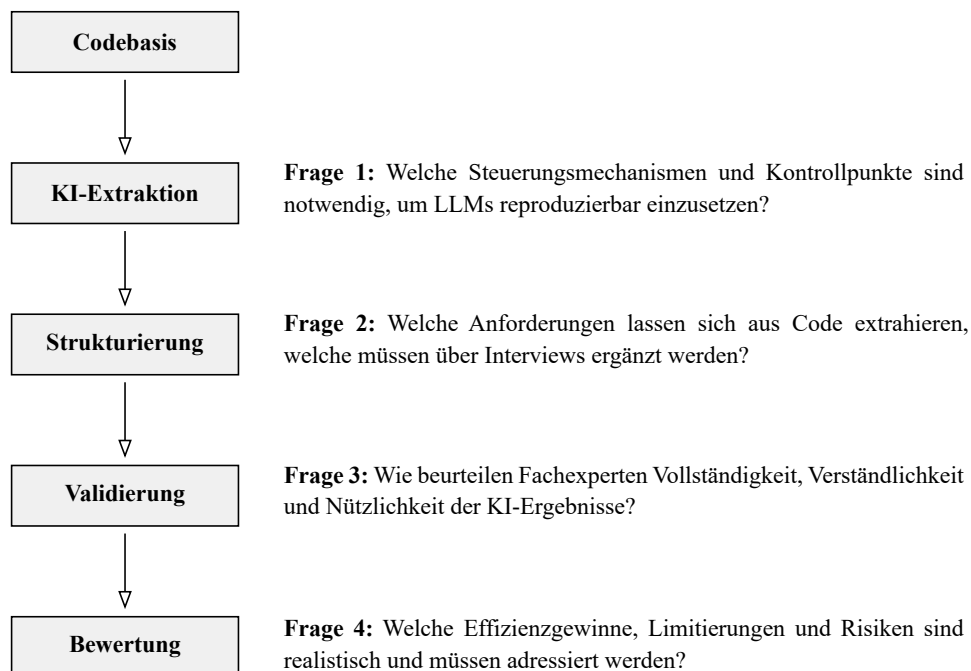


Abbildung 5: Methodisches Design im Überblick. Die vertikale Sequenz zeigt den Ablauf von der Codebasis bis zur Bewertung. Pro Phase ist die zugeordnete Forschungsleitfrage angegeben.

Der untersuchte Prozess folgt einer durchgehenden Kette von der Codebasis bis zum Requirement. Auf der Codebasis setzt eine KI-gestützte Extraktion auf. Die Ergebnisse werden in eine konsistente

Spezifikationsform überführt und durch Fachexperten validiert. Die abschließende Bewertung erfolgt mit Hilfe vordefinierter Qualitätskriterien.

Aus diesem Ablauf ergeben sich drei methodische Module, die in den folgenden Abschnitten ausgearbeitet werden. Erstens der **kontrollierte Tooling-Vergleich**. Es ist eine Versuchsreihe vorgesehen, die auf derselben Codebasis und mit demselben Grundprompt arbeitet und sich gezielt nur in einzelnen Werkzeugkomponenten unterscheidet. Die konkrete Anzahl und Zuschnitt der Versuche werden im Untersuchungsdesign festgelegt. Zweitens die **strukturierte Stakeholder-Validierung**. Jede extrahierte Anforderung soll durch Domänenexperten geprüft, anhand einer Likert-Skala bewertet und durch halbstrukturierte Interviews ergänzt werden. Drittens die **RE-Qualitätsbewertung**. Die Bewertungskriterien werden vor der Durchführung definiert, sodass eine nachträgliche Kriterienwahl ausgeschlossen ist. Die Bewertung schließt neben der klassischen RE-Qualität auch die Migrations- und Konsolidierungsperspektive ein, also die Frage, ob eine extrahierte Anforderung im Zielsystem in dieser Form erhalten bleiben oder mit anderen zusammengeführt werden sollte.

4.3 Der RRE-Prozess als Rahmen

Das in dieser Arbeit untersuchte Vorgehen folgt der in Kapitel 2 hergeleiteten siebenstufigen Methodenkette für Reverse Requirements Engineering. Die Schritte bauen aufeinander auf und decken den Weg von der ersten Abgrenzung des Untersuchungsgegenstands bis zur Validierung der gewonnenen Anforderungen ab.

- a) **Scope und Domänenabgrenzung:** Auswahl relevanter Module, Datenobjekte und Prozesse.
- b) **Artefakterhebung:** Quellcode, Konfiguration, UI-Texte, Datenbankschemata, Schnittstellenbeschreibungen und Change-Historie.
- c) **Technische Analyse:** Struktur- und Abhängigkeitsanalyse sowie Identifikation von Kernkomponenten, Regeln und Integrationspunkten.
- d) **Semantische Interpretation:** Ableitung fachlicher Aussagen aus technischen Implementierungen.
- e) **Formalisierung:** Überführung in klare, testbare Anforderungen mit Kontext, Vorbedingung und Ergebnis.
- f) **Traceability-Anreicherung:** Verknüpfung jedes Requirements mit Artefaktbelegen.
- g) **Validierung:** Review durch Fachexperten und Abgleich mit Laufzeitverhalten oder Tickets.

In dieser Arbeit werden lediglich Schritt 1 und Schritt 7 manuell durchgeführt. Die dazwischenliegenden Schritte 2 bis 6 sollen KI-gestützt automatisiert werden. Der Untersuchungsschwerpunkt liegt damit nicht nur auf der Anforderungsbeschreibung, sondern vor allem auch auf der zuverlässigen Erzeugung dieser Beschreibung durch ein LLM.

Damit das Vorgehen belastbar bleibt, sind in jedem Versuchsdurchlauf drei Eigenschaften sicherzustellen:

Belegpflicht

Jede extrahierte Anforderung muss auf ein konkretes Artefakt wie eine Datei, ein Modul, ein Datenobjekt oder einen UI-Text zurückführbar sein.

Explizite Hypothesenmarkierung

Aussagen, die nicht eindeutig aus Artefakten ableitbar sind, werden als Hypothesen markiert und gesondert validiert.

Human-in-the-loop

Die fachliche Validierung durch Domänenexperten ist nicht optional. Plausibel formulierte LLM-Ausgaben sind kein hinreichender Beweis für sachliche Korrektheit.

Mit der Festlegung der Schrittfolge, der Aufteilung zwischen Mensch und KI sowie den drei Pflicht-Eigenschaften ist der Bezugsrahmen geklärt, in dem die folgenden Abschnitte ihre Detailfragen verorten.

4.4 Auswahl des LLM

Die Wahl des konkret eingesetzten LLM bestimmt maßgeblich, welche Steuerungsmechanismen praktisch umgesetzt werden können und wie reproduzierbar die Ergebnisse erzeugt werden. Aus diesem Grund wird die Werkzeugauswahl nicht implizit vorausgesetzt, sondern entlang der fünf Kriterien aus der Zielsetzung begründet. Diese Kriterien sind Kontextfenster, Codeverständnis, Steuerbarkeit, Kosten und Datenschutz.

Zur Auswahl stehen vier aktuell verfügbare Optionen. Anthropic Claude wird über die CLI-Variante Claude Code eingebunden, die agentisches Arbeiten und MCP-Integration nativ unterstützt. OpenAI bietet mit GPT-5 und der Codex-CLI eine vergleichbare agentische Schnittstelle. Als weitere Optionen kommen Qwen 3.5 Coder mit der Möglichkeit zur lokalen Ausführung ohne Cloud-Versand sowie DeepSeek R1 als cloudbasierte Alternative infrage.

Kriterium	Claude (Claude Code)	GPT-5 (Codex)	DeepSeek R1 (Cloud)	Qwen 3.5 Coder (lokal)
Kontextfenster	bis 1 M Tokens	bis 400 k Tokens	bis 128 k Tokens	bis 256 k Tokens
Codeverständnis	hoch	hoch	hoch	hoch
Steuerbarkeit (Agenten, MCP)	nativ	über Codex-CLI	über API	über Qwen Code CLI
Kosten	API-Abrechnung	API-Abrechnung	API-Abrechnung	Eigenbetrieb
Datenschutz	Cloud-Versand	Cloud-Versand	Cloud-Versand	On-Premise

Tabelle 1: Vergleich der LLM-Optionen entlang der fünf Auswahlkriterien.

Für diese Arbeit fällt die Entscheidung auf Claude Code als primäres Werkzeug. Ausschlaggebend sind das große Kontextfenster, die native Unterstützung von Agenten und MCP-Servern sowie eine offen dokumentierte CLI für reproduzierbare Aufrufe. Die kostenseitigen und datenschutzrechtlichen Nachteile gegenüber lokalen Modellen werden durch gezielte Konfigurationsmaßnahmen adressiert. Qwen 3.5 Coder und DeepSeek R1 werden für einen optionalen LLM-Querschnitt offengehalten, sind aber nicht das primäre Werkzeug. Qwen 3.5 Coder wird dabei lokal über LM Studio als Inferenz-Runtime betrieben. Die Anbindung erfolgt über den OpenAI-kompatiblen HTTP-Endpunkt von LM Studio, an den die Qwen Code CLI und gegebenenfalls MCP-Clients adressiert werden. Damit bleibt der Versuchsaufbau ohne Cloud-Versand und ohne Übertragung kundenbezogener Codeausschnitte an externe Anbieter.

4.5 Versuchsaufbau

Der Versuchsaufbau folgt einer schrittweise aufbauenden Vergleichslogik. Versuch 1 bis 3 bilden den Kern der Reihe und werden ausschließlich auf Claude Code durchgeführt. Ausgehend von einer Baseline werden in jedem weiteren Versuch zusätzliche Werkzeugkomponenten hinzugefügt, sodass der Effekt jeder Komponente isoliert beobachtbar ist. Alle drei Versuche arbeiten auf derselben Codebasis und mit demselben Grundprompt. Variiert wird ausschließlich die Werkzeugkonfiguration. Der optionale Versuch 4 kehrt die Logik um. Die in Versuch 1 bis 3 wirksamste Konfiguration wird fixiert und auf den drei alternativen Modellen wiederholt, um modellabhängige von werkzeugabhängigen Effekten trennen zu können.

Versuch 1 (Baseline, Prompt-only)

Reine Prompt-Steuerung ohne Agentendateien und ohne externe Tools. Die Hypothese lautet, dass eine formal strukturierte Anforderungsmenge bereits ohne Spezialisierung erreichbar ist, allerdings mit begrenzter Discovery-Breite und ohne dynamische Code- oder Datenbeobachtung.

Versuch 2 (Spezialisierung über Agenten)

Wie Versuch 1, ergänzt um rollenspezialisierte Agentendateien für Stakeholder-Analyse, System-Requirements, Software-Requirements und einen ISO-29148-Orchestrator. Die Hypothese lautet, dass Spezialisierung die Strukturierungstiefe und die Normkonformität erhöht, ohne die Discovery-Breite signifikant zu verschlechtern.

Versuch 3 (Toolzugriff über MCP-Server)

Wie Versuch 2, ergänzt um strukturierten Tool-Zugriff über MCP. Vorgesehen sind drei Server für Symbol-Navigation auf Code-Ebene, für Datenbank-Inspektion auf Schema- und Datensatzebene sowie optional für GUI-Beobachtung. Die Hypothese lautet, dass strukturierter Tool-Zugriff die Discovery-Breite vergrößert und zuvor undokumentierte Use Cases sichtbar macht, allerdings zu Lasten erhöhter Steuerungskomplexität.

Versuch 4 (optional, LLM-Querschnitt)

Die in den ersten drei Versuchen wirksamste Konfiguration wird auf allen drei alternativen Modellen wiederholt: GPT-5 über die Codex-CLI, DeepSeek R1 über die Cloud-API sowie Qwen 3.5 Coder lokal über LM Studio. Wo einzelne Werkzeugkomponenten in der jeweiligen Umgebung nicht eins zu eins verfügbar sind, wird ein funktional äquivalenter Ersatz gewählt und dokumentiert. Ziel ist eine Einschätzung, in welchem Maße die in Versuch 1 bis 3 beobachteten Effekte modellabhängig oder werkzeugabhängig sind.

Versuch	Werkzeugkonfiguration	Hypothese
V1 Baseline	• Prompt-only • keine Agentendateien • keine externen Tools	• Formal strukturierte Spezifikation erreichbar • Discovery-Breite begrenzt
V2 Agenten	• Wie V1 • rollenspezialisierte Agentendateien	• Höhere Strukturierungstiefe und Normkonformität • Vergleichbare Discovery-Breite
V3 MCP-Tools	• Wie V2 • MCP-Server für Code • MCP-Server für Datenbank • optional MCP-Server für GUI	• Größere Discovery-Breite • Höhere Steuerungskomplexität
V4 (optional)	• beste Konfiguration aus V1–V3 • GPT-5 über Codex-CLI • DeepSeek R1 über Cloud-API • Qwen 3.5 Coder lokal über LM Studio	Trennung modell- gegenüber werkzeugabhängiger Effekte

Tabelle 2: Übersicht der geplanten Versuche mit Werkzeugkonfiguration und Arbeitshypothese.

4.5.1 Iteratives Vorgehen innerhalb eines Versuchs

Innerhalb jedes Versuchs läuft ein iterativer Loop ab, in dem der Prompt schrittweise verfeinert wird. Der Begriff Iteration bezeichnet in dieser Arbeit ausschließlich diesen inneren Loop. Der schrittweise Aufbau über V1, V2 und V3 wird hingegen als Versuchsreihe bezeichnet.

Der Ablauf folgt fünf Schritten. Ausgangspunkt ist ein Initial-Prompt. In Versuch 1 wird dieser zu Versuchsbeginn neu formuliert. In Versuch 2 und Versuch 3 wird stattdessen der finale Prompt des vorhergehenden Versuchs übernommen und vor dem ersten Lauf an die neu hinzukommende Werkzeugkomponente angepasst (Agentendateien beziehungsweise MCP-Server). Der Initial-Prompt wird gegen

die Codebasis ausgeführt. Anschließend begutachtet der Autor die Ausgabe stichprobenartig auf offensichtliche Lücken, formale Probleme oder fehlende Belege. Auf Basis dieser Begutachtung wird der Prompt mit dokumentiertem Änderungsgrund angepasst und erneut ausgeführt. Der Loop wiederholt sich, bis der Autor das Ergebnis als ausreichend einstuft.

Diese innere Begutachtung ersetzt nicht die Stakeholder-Validierung. Die Stakeholder-Validierung greift erst, wenn der Autor den Versuch als abgeschlossen erklärt. Ihr Urteil bezieht sich ausschließlich auf das End-Set des Versuchs. Damit fließt der Validatoren-Aufwand nicht in jede Zwischenversion ein.

Jede Prompt-Version wird im Versuchsordner mit Zeitstempel und kurzer Notiz zum Änderungsgrund abgelegt, sodass die Iterations-Historie reproduzierbar bleibt und später als Lerneffekt diskutiert werden kann. Der Ablauf ist in Abbildung 6 zusammengefasst.

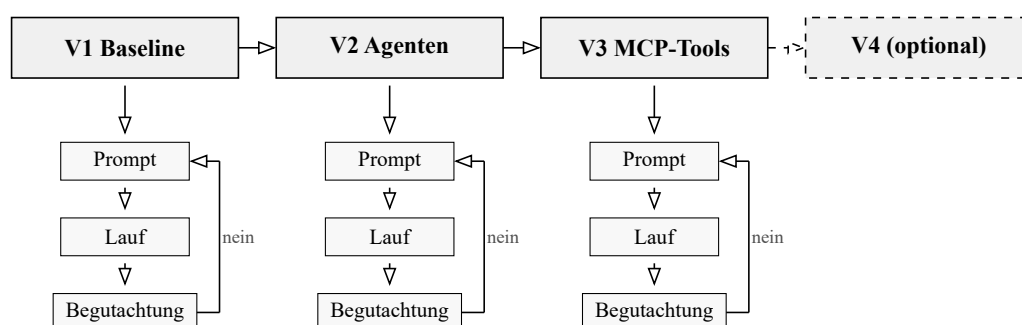


Abbildung 6: Zustandsgraph des Versuchsablaufs. Die Versuche bauen schrittweise aufeinander auf, V4 ist optional. Der finale Prompt eines Versuchs wird zu Beginn des Folgeversuchs an die neu hinzukommende Werkzeugkomponente angepasst und dient dann als Startprompt. Innerhalb jedes Versuchs verfeinert ein Iterations-Loop den Prompt: Lauf → Begutachtung durch den Autor → bei „nein,, Prompt-Anpassung, bei „ja“ Übergang zum nächsten Versuch.

Konstanten und Variablen sind in jedem Versuch klar dokumentiert. In Versuch 1 bis 3 umfassen die Konstanten Codebasis, Iterations-Vorgehen, Modellfamilie, Validierungsstichprobe und Bewertungskriterien. Variabel sind die Werkzeugkonfiguration und der Startprompt jedes Versuchs. Versuch 1 startet mit einem zu Versuchsbeginn neu formulierten Prompt. Versuch 2 und Versuch 3 übernehmen jeweils den finalen Prompt des vorhergehenden Versuchs und passen ihn zu Beginn an die neu hinzukommende Werkzeugkomponente an (Agentendateien in Versuch 2, MCP-Server in Versuch 3). Erst dieser angepasste Prompt bildet den Ausgangspunkt für den Iterations-Loop des Folgeversuchs. In Versuch 4 wird die Logik umgekehrt. Die Werkzeugkonfiguration ist die Konstante, das Modell die Variable. Damit lassen sich Unterschiede in Versuch 1 bis 3 ursächlich der Werkzeugvariation und in Versuch 4 ursächlich der Modellwahl zuordnen.

4.6 Stakeholder-Validierung als Verifikationsverfahren

Die Stakeholder-Validierung ist das zentrale Verifikationsverfahren dieser Arbeit. Sie ist nicht als nachgelagerter Schritt gedacht, sondern bildet das Maß, an dem die KI-Ergebnisse gemessen werden. Plausibel formulierte LLM-Ausgaben sind nicht hinreichend. Eine Anforderung gilt erst dann als belastbar, wenn sie durch einen Domänenexperten geprüft und bestätigt wurde.

Vorgesehen sind bis zu drei Validatoren mit jeweils mehrjähriger Erfahrung in der c-entron-Codebasis und in den fachlich abgedeckten Geschäftsprozessen. Da im Unternehmen nicht mehr für jeden Fachbereich ein dedizierter Experte verfügbar ist, kann eine vollständige modulweise Abdeckung durch bereichsspezifische Validatoren nicht garantiert werden. Die Validatorengruppe deckt die Codebasis stattdessen in Summe ab. Die Teilnehmer sind bereits identifiziert. Der Interview-Leitfaden ist in Abschnitt 10.1.1 im Anhang dokumentiert.

Für die Validierung wird pro Versuchslauf eine Zufallsstichprobe aus den extrahierten Anforderungen gezogen und auf die bis zu drei Teilnehmer verteilt. Die Stichprobengröße wird vor Versuchsbeginn festgelegt und im Versuchsprotokoll dokumentiert. Eine bereichs- oder risikoklassenspezifische Stratifizierung ist nicht vorgesehen, da die personelle Verfügbarkeit der Fachexperten eine flächendeckende Modulabdeckung nicht zulässt.

Jede Anforderung wird entlang von sechs Dimensionen bewertet:

Sachliche Korrektheit

Beschreibt die Anforderung das tatsächliche Systemverhalten?

Vollständigkeit

Sind Akteur, Vorbedingung und Ergebnis ausreichend spezifiziert?

Verständlichkeit

Lässt sich die Anforderung ohne Rückfrage interpretieren?

Redundanzfreiheit

Ist die Anforderung von anderen klar abgegrenzt?

Übernahmewürdigkeit

Soll die Anforderung im Zielsystem in ihrer Funktion erhalten bleiben, oder ist sie ein historisch gewachsener Workaround, ein Sonderfall oder eine veraltete Logik, die im Zuge der Neuimplementierung entfallen sollte?

Konsolidierungsbedarf

Bestehen andere Anforderungen, die dieselbe fachliche Funktion abbilden und im Zielsystem zu einer gemeinsamen Anforderung zusammengeführt werden sollten? Ein typisches Beispiel sind die in der bestehenden c-entron-Codebasis getrennt geführten Datensätze für Drucker („Stammblätter“) und sonstige Hardware („Assets“), die im Zielsystem zu einem konsolidierten Asset-Konzept zusammengeführt werden sollen.

Die Bewertung erfolgt auf einer fünfstufigen Likert-Skala mit definierten Ankern an den Polen, wobei 1 für „trifft nicht zu“, und 5 für „trifft voll zu“ steht.

Ergänzend zur itemweisen Bewertung werden mit den Validatoren halbstrukturierte Interviews geführt. Themen sind hierbei vor allem migrationsspezifische Risiken, der Konsolidierungsbedarf gegenüber der Zielarchitektur und die Nützlichkeit der KI-Ergebnisse im Vergleich zu einer hypothetischen manuellen Analyse.

Eine Requirement gilt im Sinne dieser Arbeit als belastbar, wenn drei Quellen sie stützen: ein KI generiertes Requirement, ein konkreter Artefakt-Beleg und eine Expertenbestätigung.

4.7 Evaluationsrahmen

Der Evaluationsrahmen wird vor der Durchführung der Versuche definiert. Damit wird eine nachträgliche Anpassung der Kriterien an die Ergebnisse ausgeschlossen. Die Bewertung orientiert sich an den drei in Abschnitt 2.2.6 hergeleiteten Qualitätsdimensionen.

Statement-Qualität

Pro einzelner Anforderung wird gemessen, ob sie eindeutig formuliert, vollständig im Satzbau, frei von unbelegten Annahmen und mit Akzeptanzkriterium oder Prüfidée versehen ist. Die Messung erfolgt über die zuvor beschriebene Likert-Skala.

Set-Qualität

Pro Versuch wird die gesamte erzeugte Anforderungsmenge als ein Set bewertet. Gemessen wird, ob sie die relevanten Prozesse und Varianten vollständig abdeckt, in sich konsistent ist und keine Doubletten enthält. Im Vordergrund steht dabei die Vollständigkeit, weil ein fehlendes Requirement im Migrationskontext zu Funktionsverlust führen kann. Die Bewertung erfolgt qualitativ durch Expertenbewertung und ergänzend durch maschinelle Konsistenzprüfungen wie doppelte IDs oder fehlende Belege.

Traceability-Qualität

Pro Beleg-Verknüpfung wird gemessen, ob der Beleg reproduzierbar auffindbar ist, etwa über Dateipfad, Methode oder SQL-Query, und ob die Ableitung vom Beleg zur Anforderung nachvollziehbar bleibt.

Ergänzend zur Qualitätsbewertung wird der Aufwand erhoben. Dazu werden während der KI-Läufe Indikatoren wie Tokenkosten und Bearbeitungsdauer protokolliert. Parallel wird grob abgeschätzt, wie viele Stunden ein erfahrener Analyst für dieselben Module ohne KI-Unterstützung gebraucht hätte. Beide Größen zusammen erlauben einen Vergleich der Größenordnung, nicht jedoch einen exakten Effizienzfaktor.

4.8 Reproduzierbarkeit und Risikomanagement

Reproduzierbarkeit und Risikomanagement sind als querschnittliche Aspekte angelegt. Sie betreffen alle Versuchsdurchläufe gleichermaßen und werden hier zusammengefasst.

Alle steuerungsrelevanten Artefakte werden versioniert vorgehalten. Hierzu zählen die verwendeten Prompts in ihrer Textfassung inklusive aller Iterationsversionen mit Zeitstempel und Änderungsgrund, die Agentendateien mit ihren Rollenbeschreibungen, die MCP-Server-Konfigurationen sowie die Angaben zu Modellversion und Kontextfenstergröße. Für lokal betriebene Modelle werden zusätzlich die Inferenz-Runtime mit Version (LM Studio), die Quantisierungsstufe des verwendeten Modell-Builds sowie weitere Sampling-Parameter dokumentiert, da diese Stellgrößen die Reproduzierbarkeit der Ausgaben spürbar beeinflussen. Jeder Versuchsordner enthält die vollständige Konfiguration als Single Source. Wo möglich, werden deterministische Einstellungen gewählt.

Da die Codebasis kundenbezogene Strukturen enthält, werden datenschutzkritische Werkzeuge bewusst eingegrenzt. MCP-Server für Datenbank-Inspektion und Symbol-Navigation werden lokal betrieben. An externe LLM-Anbieter werden nur diejenigen Codeausschnitte gesendet, die für den jeweiligen Analyseschritt notwendig sind. Personenbezogene Daten oder vollständige Datenexporte sind ausgeschlossen.

Die folgenden vier Risikokategorien werden adressiert:

Halluzinationen

Begegnet durch Belegpflicht und Stakeholder-Validierung. Jede Anforderung ohne nachvollziehbaren Beleg wird als Hypothese markiert.

Reproduzierbarkeitsverlust

Begegnet durch versionierte Prompts und deterministische Einstellungen, soweit das Modell sie unterstützt. Da die innere Iteration zudem die Gefahr eines Autoren-Bias birgt, etwa eines unbewussten

Hinsteuerns auf bestimmte Module, werden die Prompt-Änderungen pro Iteration mit Änderungsgrund dokumentiert und sind im Versuchsordner nachvollziehbar.

Domänen- und Datenbias

Begegnet durch eine Stichprobenwahl, die alle relevanten Module abdeckt und nicht nur die in der KI-Ausgabe häufig auftauchenden.

Datenschutzverletzungen

Begegnet durch On-Premise-MCP, kontrollierten Versand und Logging der externen Aufrufe.

4.9 Konkrete Konfigurationen der geplanten Versuche

Dieser Abschnitt konkretisiert die zuvor beschriebene Versuchsreihe auf Konfigurationsebene. Jeder Versuch ist durch seinen Prompt, seine Agentenliste und seine MCP-Server-Liste vollständig beschrieben. Modellversion, Kontextfenster und Temperatur werden im Versuchsordner protokolliert.

Element	V1 Baseline	V2 Agenten	V3 MCP-Tools
Modell	Claude (Claude Code)	Claude (Claude Code)	Claude (Claude Code)
Startprompt	Zu Versuchsbeginn neu formuliert	Finaler Prompt aus V1, angepasst an Agentendateien	Finaler Prompt aus V2, angepasst an MCP-Server
Agentendateien	keine	Stakeholder, System, Software, ISO-29148-Orchestrator	wie V2
MCP-Server	keine	keine	Symbol-Navigation, Datenbank-Inspektion, optional GUI-Beobachtung
Validierungsstichprobe	Zufallsstichprobe	Zufallsstichprobe	Zufallsstichprobe

Tabelle 3: Detail-Konfiguration der drei Kernversuche.

Die Versuchsordner-Struktur folgt einer einheitlichen Konvention. Pro Versuch existiert ein Unterordner mit den Konfigurationsartefakten, ein Eingangsprotokoll mit Modell- und Werkzeugangaben, ein Ergebnis-Unterordner sowie eine Verlaufsdocumentation für die Validierungsschritte. Damit ist jeder Versuch eigenständig reproduzierbar.

5 Durchführung und Ergebnisse

6 Evaluation (ca. 12 Seiten)

7 Diskussion

7.1 Interpretation der Ergebnisse

7.2 Chancen und Grenzen

7.3 Implikationen fuer Forschung und Praxis

8 Fazit und Ausblick

9 Literaturverzeichnis (ca. 3 Seiten)

Bibliografie

- Anthropic. (2026,). *Claude Code Documentation: MCP*. <https://code.claude.com/docs/en/mcp>
- B, L., Wiegers, K., & Ebert, C. (2001). The top risk of requirements engineering. *IEEE Software*, 18(6), 62–63. <https://doi.org/10.1109/52.965804>
- Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? 🦜. *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 610–623. <https://doi.org/10.1145/3442188.3445922>
- Bisbal, J., Lawless, D., Wu, B., & Grimson, J. (1999). Legacy information systems: issues and directions. *IEEE Software*, 16(5), 103–111. <https://doi.org/10.1109/52.795108>
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer. <https://link.springer.com/book/9780387310732>
- Chikofsky, E., & Cross, J. (1990). Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1), 13–17. <https://doi.org/10.1109/52.43044>
- Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). Large Language Models for Software Engineering: Survey and Open Problems. *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 31–53. <https://doi.org/10.1109/icse-fose59343.2023.00008>
- Fantechi, A., Gnesi, S., Passaro, L., & Semini, L. (2023). Inconsistency Detection in Natural Language Requirements using ChatGPT: a Preliminary Evaluation. *2023 IEEE 31st International Requirements Engineering Conference (RE)*, 335–340. <https://doi.org/10.1109/re57278.2023.00045>
- Glinz, M. (2007). On Non-Functional Requirements. *15th IEEE International Requirements Engineering Conference (RE 2007)*, 21–26. <https://doi.org/10.1109/re.2007.45>
- Glinz, M. (2008). A Risk-Based, Value-Oriented Approach to Quality Requirements. *IEEE Software*, 25(2), 34–41. <https://doi.org/10.1109/ms.2008.31>
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. <https://www.deeplearningbook.org/>
- Gotel, O., & Finkelstein, C. (o. J.). An analysis of the requirements traceability problem. *Proceedings of IEEE International Conference on Requirements Engineering*, 94–101. <https://doi.org/10.1109/icre.1994.292398>

- Hassani, S. (2024). Enhancing Legal Compliance and Regulation Analysis with Large Language Models. *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, 507–511. <https://doi.org/10.1109/re59067.2024.00065>
- Hemmat, A., Sharbaf, M., Kolahdouz-Rahimi, S., Lano, K., & Tehrani, S. Y. (2025). Research directions for using LLM in software requirement engineering: a systematic review. *Frontiers in Computer Science*, 7. <https://doi.org/10.3389/fcomp.2025.1519437>
- Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2023,). *Large Language Models for Software Engineering: A Systematic Literature Review*. <https://arxiv.org/abs/2308.10620v6>
- IEEE. (1998,). *IEEE Std 830-1998: Recommended Practice for Software Requirements Specifications*. <https://standards.ieee.org/standard/830-1998.html>
- ISO/IEC. (2011,). *ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>
- ISO/IEC/IEEE. (2018,). *ISO/IEC/IEEE 29148:2018 Systems and software engineering — Life cycle processes — Requirements engineering*. <https://www.iso.org/standard/72089.html>
- Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., & Fung, P. (2023). Survey of Hallucination in Natural Language Generation. *ACM Computing Surveys*, 55(12), 1–38. <https://doi.org/10.1145/3571730>
- Kotonya, G., & Sommerville, I. (1996). Requirements engineering with viewpoints. *Software Engineering Journal*, 11(1), 5. <https://doi.org/10.1049/sej.1996.0002>
- Kratzke, N., & Quint, P.-C. (2017). Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. *Journal of Systems and Software*, 126, 1–16. <https://doi.org/10.1016/j.jss.2017.01.001>
- Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., & Kiela, D. (2020,). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. <https://arxiv.org/abs/2005.11401v4>
- Luitel, D., Hassani, S., & Sabetzadeh, M. (2024). Improving requirements completeness: automated assistance through large language models. *Requirements Engineering*, 29(1), 73–95. <https://doi.org/10.1007/s00766-024-00416-3>
- Marczak-Czajka, A., & Cleland-Huang, J. (2023). Using ChatGPT to Generate Human-Value User Stories as Inspirational Triggers. *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, 52–61. <https://doi.org/10.1109/rew57809.2023.00016>
- Marques, N., Silva, R. R., & Bernardino, J. (2024). Using ChatGPT in Software Requirements Engineering: A Comprehensive Review. *Future Internet*, 16(6), 180. <https://doi.org/10.3390/fi16060180>
- Norheim, J. J., & Rebertisch, E. (2024). Structuring Natural Language Requirements with Large Language Models. *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*, 68–71. <https://doi.org/10.1109/rew61692.2024.00013>
- Nouri, A., Cabrero-Daniel, B., Törner, F., Sivencrona, H., & Berger, C. (2024). Engineering Safety Requirements for Autonomous Driving with Large Language Models. *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, 218–228. <https://doi.org/10.1109/re59067.2024.00029>

- Okamoto, R., & Kusumoto, S. (2025). Towards the Automatic Restructuring of Software Requirements Specifications to Conform to Standards Using Large Language Models. *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, 467–475. <https://doi.org/10.1109/re63999.2025.00056>
- OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Alteschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., ... Zoph, B. (2023,). *GPT-4 Technical Report*. <https://arxiv.org/abs/2303.08774v6>
- Pahl, C., & Jamshidi, P. (2016). Microservices: A Systematic Mapping Study. *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, 137–146. <https://doi.org/10.5220/0005785501370146>
- Pohl, K. (2010). *Requirements Engineering*. Springer Berlin Heidelberg. <https://doi.org/10.1007/978-3-642-12578-2>
- Salem, N., Hudaib, A., Al-Tarawneh, K., Salem, H., Tareef, A., Salloum, H., & Mazzara, M. (2024). A survey on the application of large language models in software engineering. *Computer Research and Modeling*, 16(7), 1715–1726. <https://doi.org/10.20537/2076-7633-2024-16-7-1715-1726>
- Sneed, H. (1995). Planning the reengineering of legacy systems. *IEEE Software*, 12(1), 24–34. <https://doi.org/10.1109/52.363168>
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017,). *Attention Is All You Need*. <https://arxiv.org/abs/1706.03762v7>
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022,). *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. <https://arxiv.org/abs/2201.11903v6>
- Yu, Y., Mylopoulos, J., Wang, Y., Liaskos, S., Lapouchnian, A., Zou, Y., Littou, M., & Leite, J. (o. J.). RETR: Reverse Engineering to Requirements. *12th Working Conference on Reverse Engineering (WCRE '05)*, 234. <https://doi.org/10.1109/wcre.2005.27>
- Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S., & Chen, Z. (2023,). *A Survey on Large Language Models for Software Engineering*. <https://arxiv.org/abs/2312.15223v2>

10 Anhang (ca. 6 Seiten)

10.1 Interviewleitfäden

10.1.1 Stakeholder-Validierung der KI-extrahierten Anforderungen

Ausarbeitung folgt.

10.2 Zusätzliches Datenmaterial

10.3 Konfigurationsdetails des Prototyps