Themensteller: Prof. Dr. Henning Femmer

vorgelegt von: Ivan Esau

Matrikelnummer: 30323102  
Hermann-Löns Str.7  
58708 Menden  
esau.ivan@fh-swf.de

Abgabetermin: 2025-10-10

**Simulationsumgebung zur automatischen Durchführung von KI-basierten Software-Engineering-Simulationen in GitLab**

Bachelorarbeit

Sommersemester 2025

Inhaltsverzeichnis

[Abbildungsverzeichnis V](#_Toc210642382)

[Tabellenverzeichnis V](#_Toc210642383)

[Abkürzungsverzeichnis VI](#_Toc210642384)

[1 Einleitung 1](#_Toc210642385)

[1.1 Ausgangslage & Motivation 1](#_Toc210642386)

[1.2 Problemstellung & Forschungslücke 2](#_Toc210642387)

[1.3 Zielsetzung & Forschungsfragen 3](#_Toc210642388)

[1.4 Vorgehensweise 4](#_Toc210642389)

[1.5 Aufbau der Arbeit 5](#_Toc210642390)

[2 Theoretische Grundlagen 7](#_Toc210642391)

[2.1 Software-Engineering-Workflow in GitLab 7](#_Toc210642392)

[2.1.1 Merge-Requests 7](#_Toc210642393)

[2.1.2 CI/CD-Pipelines 8](#_Toc210642394)

[2.2 Requirements-Engineering 9](#_Toc210642395)

[2.2.1 Requirements-Quality: ISO/IEC/IEEE 29148:2018 9](#_Toc210642396)

[2.2.2 Requirements Engineering im Kontext von KI 10](#_Toc210642397)

[2.3 LLM-Grundlagen 11](#_Toc210642398)

[2.3.1 Funktionsweise 11](#_Toc210642399)

[2.3.2 Prompt-Engineering 12](#_Toc210642400)

[2.3.3 Function-Calling 13](#_Toc210642401)

[2.4 Agentsysteme 14](#_Toc210642402)

[2.4.1 Grundlagen von Agentensystemen 14](#_Toc210642403)

[2.4.2 Klassische Agentensysteme 15](#_Toc210642404)

[2.4.3 Agenten im Kontext von LLMs 16](#_Toc210642405)

[2.4.4 Vergleich: Klassische vs. LLM-basierte Agenten 18](#_Toc210642406)

[2.4.5 Supervisor- und Orchestrator-Pattern 19](#_Toc210642407)

[2.4.6 Herausforderungen 20](#_Toc210642408)

[2.5 Model Context Protocol 21](#_Toc210642409)

[3 Forschungsdesign & Methodik 23](#_Toc210642410)

[3.1 Design-Science-Research 23](#_Toc210642411)

[3.2 Untersuchungsmodell & Hypothesen 24](#_Toc210642412)

[3.2.1 Untersuchungsmodell 24](#_Toc210642413)

[3.2.2 Hypothesen 25](#_Toc210642414)

[3.3 Versuchsplanung 26](#_Toc210642415)

[3.4 Messkonzept & Instrumentierung 27](#_Toc210642416)

[3.4.1 Erfolgsmetriken 27](#_Toc210642417)

[3.4.2 Berechnungen 28](#_Toc210642418)

[3.5 Auswertungsverfahren 29](#_Toc210642419)

[4 Systementwurf 31](#_Toc210642420)

[4.1 Anforderungen 31](#_Toc210642421)

[4.1.1 Funktionale Anforderungen 31](#_Toc210642422)

[4.1.2 Nicht-funktionale Anforderungen 33](#_Toc210642423)

[4.2 Lösungskonzept 34](#_Toc210642424)

[4.3 Architekturkonzept 35](#_Toc210642425)

[4.4 Multi-Agenten-Orchestrierung 36](#_Toc210642426)

[4.4.1 Supervisor-Pattern als Koordinationsmodell 36](#_Toc210642427)

[4.4.2 ReAct-Agent 37](#_Toc210642428)

[4.4.3 Spezialisierte Agentenrollen 38](#_Toc210642429)

[4.4.4 Kommunikations- und Koordinationskonzept 40](#_Toc210642430)

[4.5 Prompt-Design 41](#_Toc210642431)

[4.6 Workflow-Design 42](#_Toc210642432)

[4.6.1 Initialisierungsphase 42](#_Toc210642433)

[4.6.2 Planungphase 42](#_Toc210642434)

[4.6.3 Implementierungsphase 42](#_Toc210642435)

[4.6.4 Abschluss 42](#_Toc210642436)

[4.7 Datenfluss 43](#_Toc210642437)

[4.8 Messkonzept 44](#_Toc210642438)

[4.8.1 Erfolgsmetriken 44](#_Toc210642439)

[4.8.2 Prozessmetriken 44](#_Toc210642440)

[4.8.3 Metriken-Speicherung-Konzept 45](#_Toc210642441)

[5 Implementierung 46](#_Toc210642442)

[5.1 Entwicklungszyklen 46](#_Toc210642443)

[5.2 Technologie-Stack 46](#_Toc210642444)

[5.3 Implementierungsschwerpunkt: Prompt-Engineering 49](#_Toc210642445)

[5.3.1 Herausforderung: LLM-Arbeit mit GitLab gesteuert durch Prompts 49](#_Toc210642446)

[5.3.2 Iterativer Prompt-Engineering-Prozess 50](#_Toc210642447)

[5.3.3 Prompt-Template-Struktur 52](#_Toc210642448)

[5.4 Implementierungsschwerpunkt: Orchestrierung 53](#_Toc210642449)

[5.4.1 Sequentielle Koordination 54](#_Toc210642450)

[5.5 Ergebnis der Implementation 54](#_Toc210642451)

[6 Evaluation & Experimente 56](#_Toc210642452)

[6.1 Planung & Aufbau der Experimente 56](#_Toc210642453)

[6.1.1 Projektauswahl 57](#_Toc210642454)

[6.1.2 Pipeline aufbau 58](#_Toc210642455)

[6.1.3 Projekt-Template A 59](#_Toc210642456)

[6.1.4 Projekt-Template B 60](#_Toc210642457)

[6.2 Durchführung 61](#_Toc210642458)

[6.3 Bewertung und Analyse 62](#_Toc210642459)

[6.3.1 Auswertung: Varianz der Projekte Durchläufe(FF.2) 63](#_Toc210642460)

[6.3.2 Gegenüberstellung: Projekt Ergebnisse (FF.1) 63](#_Toc210642461)

[6.3.3 Zusätzliche Beobachtungen 63](#_Toc210642462)

[7 Diskussion 64](#_Toc210642463)

[7.1 Interpretation 64](#_Toc210642464)

[7.2 Limitationen 65](#_Toc210642465)

[7.3 Herausforderungen 66](#_Toc210642466)

[7.3.1 Technische Herausforderungen 66](#_Toc210642467)

[7.3.2 Organisatorische und methodische Herausforderungen 67](#_Toc210642468)

[7.4 Ausblick 67](#_Toc210642469)

[8 Fazit 69](#_Toc210642470)

[Anhang 70](#_Toc210642471)

[9 References 73](#_Toc210642472)

# Abbildungsverzeichnis

[Abbildung 1 Transofrmoer Architektur 11](#_Toc210521095)

[Abbildung 2 Chain of Thought concepts 12](#_Toc210521096)

[Abbildung 3 function Calling 13](#_Toc210521097)

[Abbildung 4 BDI Aktivitätsdiagramm [40] 15](#_Toc210521098)

[Abbildung 5 MCP - Architektur 21](#_Toc210521099)

[Abbildung 6 Workflow Phasen 35](#_Toc210521100)

[Abbildung 7 Schichtenarchitektur 36](#_Toc210521101)

[Abbildung 8 ReAct-Pattern 38](#_Toc210521102)

[Abbildung 9 Sequentieller Issue Prozess 41](#_Toc210521103)

# Tabellenverzeichnis

[Tabelle 1 Agenten Vergleich 18](#_Toc210521104)

[Tabelle 2 Multi-Pattern Success-Marker-System 42](#_Toc210521105)

# Abkürzungsverzeichnis

KI Künstliche Intelligenz

LLM Large Language Model

MCP Model Context Protocol

CI/DC Continues ….

MR Merge-Request

# Einleitung

In diesem Kapitel wird eine konkrete Einführung in das Thema der Bachelorarbeit gegeben und der Grundstein für das aufzubauende System gelegt, durch Darstellung der Ausgangslage, erklärung der Problemstellung und einer Zielsetzung.

## Ausgangslage & Motivation

In den letzten Jahren haben Large Language Models (LLMs) die Softwaretechnik tiefgreifend verändert. [1, 2] Während frühe Arbeiten auf einfache Code-Vervollständigung fokussierten [3, 4], ermöglichen heutige Systeme autonome Softwareentwicklungs-Agenten, die komplexe Aufgaben wie Bugfixing, Testing oder Dokumentation durchführen. [5–7] Diese Entwicklung geht über reine Codegenerierung hinaus und integriert Planung, Interaktion und Tool-Nutzung in sogenannten Agentischen-Workflows. [8] Plattformen wie GitLab bieten mit Merge Requests (MRs) und CI/CD-Pipelines eine realitätsnahe Infrastruktur, in der solche Agentischen-Systeme evaluiert werden können. [9] Benchmarks wie SWE-bench und SWE-bench-Live messen die Patch-Korrektheit auf statischen Repository-Snapshots [10], sind jedoch nicht in der Lage, End-to-End-Prozessmetriken (z. B. Build-Erfolg, MR-Integration, Durchlaufzeiten) zu erfassen. [5, 11]

Parallel dazu rücken Fragen nach Requirements-Qualität stärker in den Vordergrund. Die Norm ISO/IEC/IEEE 29148:2018 betont Kriterien wie Eindeutigkeit, Testbarkeit und Umsetzbarkeit von Anforderungen. [12] Neuere systematische Reviews zeigen, dass LLMs zunehmend in der Requirements Engineering (RE)-Praxis eingesetzt werden, um User Stories zu analysieren und präziser zu formulieren. [2, 13] Dennoch ist unklar, wie sich die Qualität von Anforderungen auf den Erfolg von LLM-Agenten in SE-Workflows auswirkt.

Die wachsende Komplexität erfordert zudem Interoperabilität und Portabilität. Mit dem Model Context Protocol (MCP) ist seit 2025 ein offener Standard verfügbar, der die Anbindung von Tools und Datenquellen für LLM-Agenten vereinheitlicht. [14, 15] Dies bietet erstmals die Möglichkeit, reproduzierbare E2E-Simulationen über Modell- und Providergrenzen hinweg zu realisieren.

## Problemstellung & Forschungslücke

Trotz beeindruckender Fortschritte bestehen wesentliche Lücken in der Evaluationspraxis:

1. **Begrenzte Benchmarks**: Aktuelle Studien zeigen, dass Benchmarks wie SWE-bench nur eingeschränkte Realitätsnähe besitzen, da sie oft Tests mit unvollständiger Coverage nutzen und Prozessmetriken ignorieren. [5, 11]
2. **Fragilität von Ergebnissen**: LLM- und Agenten-Leistungen variieren stark je nach Prompt, Scaffold und Tool-Zugriff. Kleine Änderungen führen zu großen Leistungsdifferenzen. [8, 16]
3. **Unzureichende Berücksichtigung von RE-Qualität**: Während Frameworks wie INVEST und QUS etabliert sind, fehlt es an empirischer Evidenz, wie Anforderungen die Performanz von LLM-Agenten in CI/CD beeinflussen. [13]
4. **Fehlende MCP-Evaluationsstudien**: Obwohl MCP die Standardisierung von Tool-Schnittstellen verspricht, existieren noch keine systematischen Untersuchungen seiner Rolle in realistischen GitLab-Simulationen. [14]

Die **aktuelle Forschung** zur Evaluation von LLMs hebt die Dringlichkeit hervor, reproduzierbare und robuste Verfahren zu entwickeln. [11, 16] Gleichzeitig wird gefordert, Studien transparenter zu dokumentieren (Prompts, Modelle, Konfigurationen), um Replizierbarkeit sicherzustellen. [16]

## Zielsetzung & Forschungsfragen

Ziel dieser Arbeit ist die Konzeption und prototypische Umsetzung von einer Simulationsumgebung, die vollständige Software-Engineering-Workflows in GitLab Ende-zu-Ende orchestriert von der Branch-Erstellung über Merge Requests bis hin zur Ausführung von CI/CD-Pipelines und dem abschließenden Merge. Mit dieser Umgebung sollen sowohl Ergebnismetriken (z. B. korrekte Problemlösung, MR-/Pipeline-Erfolg) als auch Prozessmetriken (z. B. Durchlaufzeiten, Fehlerraten) unter reproduzierbaren Bedingungen erhoben werden können.

Die Entwicklung verfolgt drei zentrale Zielsetzungen:

1. **Reproduzierbare E2E-Simulationen:** Aufbau einer kontrollierten Umgebung, in der GitLab-Workflows realitätsnah, aber reproduzierbar durchlaufen werden können. [10] So lassen sich konsistente A/B-Vergleiche zwischen Modellen und Konfigurationen realisieren. [16]
2. **Mehragentige Workflow-Orchestrierung:** Integration spezialisierter Agentenrollen (z. B. Planning, Coding, Testing, Review), die über ein Supervisor-Pattern koordiniert werden. [8] Damit lassen sich komplexe Interaktionsmuster abbilden, wie sie in aktuellen agentischen Software-Engineering-Systemen zunehmend auftreten. [7]
3. **Standardisierte Tool- und Datenanbindung:** Nutzung des **Model Context Protocols (MCP)**, das seit 2025 als offener Standard für Tool- und Datenintegration vorliegt. MCP ermöglicht eine konsistente, portierbare und auditierbare Anbindung von GitLab-Artefakten. [14, 15]

Vor diesem Hintergrund adressiert die Arbeit zwei Forschungsfragen (FF):

* **FF.1:** Wie wirkt sich die syntaktische Qualität von Anforderungsformulierungen (z. B. klare Struktur, konsistente Satzmuster, eindeutige Ausdrucksweise) auf die Effektivität von LLM-Agenten in GitLab-Workflows aus?  
  → Hierbei wird untersucht, ob präzise und strukturiert formulierte Anforderungen (im Sinne von ISO/IEC/IEEE 29148 und QUS-Kriterien) zu messbar besseren Ergebnissen bei Problemlösungen und Prozessmetriken führen. [17]
* **FF.2:** Welche Varianz zeigt das System bei wiederholten Durchläufen mit identischen Parametern und Konfigurationen?  
  → Damit wird die Robustheit und Reproduzierbarkeit der Ergebnisse in einem realitätsnahen, aber kontrollierten Evaluationsrahmen analysiert.

Die Beantwortung erfolgt empirisch anhand von E2E-Simulationen. Die Ergebnisse sollen eine fundierte Basis schaffen, um die Rolle der **syntaktischen Anforderungsqualität** im Zusammenspiel mit agentischen LLM-Systemen besser zu verstehen und gleichzeitig die Reproduzierbarkeit solcher Simulationen zu evaluieren.

## Vorgehensweise

Methodisch ist die Arbeit als **Design Science Research (DSR)** verankert. DSR verfolgt das Ziel, durch die Entwicklung von Artefakten praxisrelevante Probleme zu adressieren und dabei gleichzeitig wissenschaftliche Strenge sicherzustellen. Dieser Ansatz ist besonders geeignet für Arbeiten im Schnittfeld von Software Engineering und Künstlicher Intelligenz, da hier sowohl technologische Innovationen als auch methodisch fundierte Evaluationsstrategien gefordert sind.

Das Vorgehen folgt dem etablierten Prozessmodell von **Peffers et al. [18]**, das sechs Schritte umfasst:

1. **Problemidentifikation**: Ausgangspunkt ist die Beobachtung, dass aktuelle Benchmarks für LLMs im Software Engineering (z. B. SWE-bench) primär statische Repositorien und Testfälle abbilden, während Prozessmetriken in End-to-End-(E2E)-Workflows unzureichend berücksichtigt werden.
2. **Zieldefinition**: Ziel ist die Entwicklung einer Simulationsumgebung, die vollständige GitLab-Workflows orchestriert und dabei sowohl Ergebnis- als auch Prozessmetriken systematisch erfasst.
3. **Design und Entwicklung**: Entwurf und prototypische Umsetzung eines mehragentigen Systems, das auf modularen Rollen basiert und über standardisierte Schnittstellen mit GitLab interagiert.
4. **Demonstration**: Durchführung exemplarischer Simulationen, die den E2E-Charakter der Umgebung verdeutlichen.
5. **Evaluation**: Empirische Überprüfung anhand von wiederholten Durchläufen, um sowohl die Wirkung der syntaktischen Anforderungsqualität (FF.1) als auch die Varianz zwischen identischen Runs (FF.2) zu analysieren.
6. **Kommunikation**: Dokumentation und Aufbereitung der Ergebnisse in wissenschaftlicher Form, unter Einhaltung aktueller Empfehlungen zur Transparenz und Reproduzierbarkeit.

## Aufbau der Arbeit

Der Aufbau der Arbeit folgt einer klaren Argumentationslinie, die von den theoretischen Grundlagen über den Entwurf und die Implementierung des Artefakts bis hin zur Evaluation und abschließenden Reflexion reicht. Nach der Einleitung werden in Kapitel 2 zunächst die theoretischen Grundlagen dargestellt, die den Untersuchungsrahmen dieser Arbeit bilden. Dazu gehören der Software-Engineering-Workflow in GitLab mit seinen zentralen Elementen wie Merge Requests und CI/CD-Pipelines, die Qualitätsbegriffe des Requirements Engineering auf Basis der ISO/IEC/IEEE 29148 sowie ergänzender Konzepte wie der INVEST-Heuristik, die grundlegenden Prinzipien großer Sprachmodelle einschließlich Prompt-Engineering, Function-Calling und agentischer Architekturen sowie das Model Context Protocol (MCP) als Standard zur Integration von Tools und Datenquellen.

Kapitel 3 legt anschließend das Forschungsdesign und die Methodik offen. Im Mittelpunkt steht dabei die Einbettung der Arbeit in das Paradigma des Design-Science-Research, ergänzt durch die Formulierung des Untersuchungsmodells und der Hypothesen, die Planung der Versuche sowie die Definition von Mess- und Auswertungsverfahren.

Darauf aufbauend wird in Kapitel 4 der Systementwurf des Artefakts beschrieben. Hier werden die Anforderungen und die gewählte Architektur erläutert, die auf einem Supervisor-Pattern mit mehreren Agentenrollen basiert. Zudem werden die Abstraktionsschicht zur Nutzung unterschiedlicher LLM-Provider, die MCP-gestützte Integration in GitLab sowie die entwickelten Prompts und der Ablauf der Simulationen dargestellt.

Die praktische Umsetzung des Artefakts wird in Kapitel 5 dokumentiert. Dieses Kapitel beschreibt den eingesetzten Technologiestack, die zentralen Module und Schnittstellen sowie die Maßnahmen zur Sicherstellung von Konfigurationstreue und Reproduzierbarkeit, etwa durch Containerisierung und Infrastructure-as-Code.

Kapitel 6 widmet sich der Evaluation und den Experimenten. Hier werden zunächst die Datengrundlage und die Szenarien vorgestellt, anschließend das Design der Experimente erläutert und schließlich die Ergebnisse zu den Forschungsfragen präsentiert. Eine zusammenfassende Betrachtung der Befunde schließt dieses Kapitel ab.

In Kapitel 7 werden die Resultate kritisch diskutiert. Dabei erfolgt eine Interpretation der Befunde im Lichte bestehender Forschung, es werden Limitationen des entwickelten Ansatzes benannt und mögliche Perspektiven für weiterführende Arbeiten aufgezeigt.

Kapitel 8 bildet den Abschluss der Arbeit. Es fasst die zentralen Erkenntnisse zusammen, beantwortet die Forschungsfragen in prägnanter Form und gibt einen Ausblick auf zukünftige Entwicklungen im Bereich agentischer LLM-Systeme für Software-Engineering-Workflows. Abgerundet wird die Arbeit durch den Anhang mit ergänzenden Materialien sowie durch das Literaturverzeichnis in Kapitel 9, das die verwendeten Quellen im IEEE-Zitierstil aufführt.

# Theoretische Grundlagen

Einleitung in die aufkommenden Themengebiete....

## Software-Engineering-Workflow in GitLab

GitLab hat sich als führende Plattform für kollaboratives Software-Engineering etabliert und bietet ein integriertes DevOps-Ökosystem, das alle Phasen des Software-Lebenszyklus von Quellcodeverwaltung über CI/CD bis hin zum Monitoring in einer einzigen Anwendung vereint. [19]

Besonders im Kontext moderner, agiler Entwicklungsmethoden wie DevOps oder GitOps sind GitLab-Funktionalitäten wie Merge Requests (MRs) und CI/CD-Pipelines zentrale Bestandteile für Qualitätssicherung, Nachvollziehbarkeit und Automatisierung. [20, 21]

GitLab kombiniert durchgängige Automatisierung mit Versionskontrolle und Review-Mechanismen und eignet sich dadurch sowohl für industrielle Anwendungen als auch für empirische Studien zu Softwareentwicklungsprozessen. [16, 20] Die folgenden Unterkapitel beleuchten die Rolle von Merge Requests und CI/CD-Pipelines im Detail.

### Merge-Requests

Merge Requests (MRs) in GitLab sind mehr als nur ein Mechanismus zur Integration von Code in den Hauptentwicklungszweig. Sie fungieren als zentrale Schnittstelle zur technischen, organisatorischen und qualitativen Koordination von Änderungen. Ein MR enthält typischerweise Code-Diffs, Diskussionen, Reviewer-Kommentare, Testresultate und Statusinformationen über CI-Ausführungen. [22]

Hilton et al. zeigten in einer groß angelegten Studie mit OSS-Projekten, dass Plattformen mit MR-gestützten Workflows und integrierter CI deutlich geringere Raten fehlerhafter Builds aufweisen. [9] Die systematische Nutzung von MRs führt zu einer erhöhten Review-Qualität und besseren Nachvollziehbarkeit, da Änderungen explizit kommentiert und in einen nachvollziehbaren Freigabeprozess eingebettet werden können. [23]

Darüber hinaus fördern MRs die kollaborative Wissensverteilung im Team. Mäntylä et al. argumentieren, dass strukturierte Code-Reviews in GitLab nicht nur die technische Codequalität verbessern, sondern auch die Kommunikation über Designentscheidungen, Best Practices und Sicherheitsaspekte dokumentieren. [24] Dies unterstützt insbesondere große verteilte Entwicklungsteams beim Wissensaufbau und bei der Vermeidung redundanter Fehler.

Durch die Verbindung technischer Integration mit sozialem Review und automatisierter Qualitätssicherung bilden MRs das organisatorische Rückgrat für transparente und robuste Entwicklungsprozesse in GitLab. [9, 21]

### CI/CD-Pipelines

CI/CD-Pipelines (Continuous Integration / Continuous Delivery) sind das Herzstück moderner Softwareautomatisierung. In GitLab werden sie in der .gitlab-ci.yml-Datei definiert und durch Events wie Pushes, Merge Requests oder Schedules ausgelöst. Sie bestehen aus einer Sequenz von Stages (z. B. Build, Test, Deploy), die von GitLab Runnern parallel oder sequenziell ausgeführt werden. [19, 20]

Donca et al. heben hervor, dass gut konstruierte CI/CD-Pipelines maßgeblich zur Standardisierung, Skalierbarkeit und Zuverlässigkeit in agilen Softwareprojekten beitragen. [20] Die Integration mehrstufiger Testverfahren (z. B. Unit, Integration, Performance) in automatisierte Pipelines erlaubt frühzeitige Fehlererkennung, erhöht die Produktqualität und reduziert Deployment-Risiken. [20, 21]

Zhao et al. belegen empirisch, dass kontinuierliche Integration (CI) in Open-Source-Projekten zu kürzeren Release-Zyklen, höherer Änderungsgeschwindigkeit und geringeren Fehlerraten führt – insbesondere, wenn Tests frühzeitig und automatisiert durchgeführt werden. [25] Gleichzeitig zeigen sie, dass Tests der häufigste Auslöser für fehlschlagende Builds sind, was die Relevanz frühzeitiger Qualitätssicherung unterstreicht. [25, 26]

Ein zentraler Vorteil von GitLab liegt in der engen Verzahnung von CI/CD mit weiteren Funktionen wie MRs, Code-Reviews, Issue-Tracking und Security-Checks. Dadurch entsteht ein durchgängiger, auditierbarer Workflow, der sich ideal für reproduzierbare Softwareentwicklungsprozesse eignet. [16, 20, 22]

## Requirements-Engineering

Das Requirements Engineering (RE) umfasst die systematische Erhebung, Analyse, Dokumentation, Validierung und Verwaltung von Anforderungen. Es stellt sicher, dass Softwareprodukte die Bedürfnisse der Stakeholder erfüllen und gleichzeitig technisch realisierbar sind. In der Literatur wird RE als „kritischer Erfolgsfaktor“ für den gesamten Software-Lebenszyklus bezeichnet, da Fehler in Anforderungen die häufigste Ursache für Projektprobleme und Nacharbeiten darstellen.

RE integriert sowohl technische Aspekte (z. B. formale Dokumentation, Nachverfolgbarkeit) als auch soziale Aspekte (z. B. Stakeholder-Kommunikation, Wissensmanagement). Internationale Standards wie ISO/IEC/IEEE 29148:2018 liefern ein Rahmenwerk für die Definition und Bewertung von Anforderungen. [12]

### Requirements-Quality: ISO/IEC/IEEE 29148:2018

Die Norm ISO/IEC/IEEE 29148:2018 definiert Qualitätsmerkmale für Anforderungen, darunter Eindeutigkeit, Vollständigkeit, Konsistenz, Umsetzbarkeit, Testbarkeit und Nachverfolgbarkeit. [12] Diese Kriterien bilden die Basis für die Beurteilung, ob Anforderungen für den Softwareentwicklungsprozess geeignet sind.

Empirische Studien zeigen jedoch, dass die praktische Anwendung dieser Kriterien herausfordernd ist. Kummler und Fromm identifizieren Schwierigkeiten bei der objektiven Bewertung von Anforderungsqualität aufgrund von Mehrdeutigkeit in der natürlichen Sprache. [27] Boyarchuk et al. schlagen strukturorientierte Prüfmethoden vor, um Anforderungen anhand der Norm systematisch zu validieren. [28]

Atoum et al. betonen in einer systematischen Literaturübersicht, dass es an empirischer Evidenz fehlt, wie stark einzelne ISO-29148-Kriterien tatsächlich die Projektergebnisse beeinflussen. [29] Montgomery et al. weisen zudem auf die wachsende Forschung zu automatisierten Ansätzen hin [30]: So nutzen Buchmann und Hauser Text Mining, um Qualitätskriterien in natürlichsprachlichen Anforderungen zu erkennen [31], während Sahu und Rai ein NLP-basiertes Framework zur Überprüfung der ISO-29148-Konformität vorstellen. [32]

Damit bleibt die empirische Validierung und Automatisierung von Qualitätsprüfungen eine zentrale Herausforderung und Forschungsrichtung im Requirements Engineering.

### Requirements Engineering im Kontext von KI

Mit dem Aufkommen von Large Language Models (LLMs) und KI-basierten Analysetools erlebt das Requirements Engineering derzeit einen tiefgreifenden Wandel. Während klassische Methoden auf manueller Analyse und formaler Dokumentation beruhen, eröffnen KI-Verfahren neue Möglichkeiten zur automatisierten Anforderungserhebung, -analyse und -validierung. [2]

Hemmat et al. zeigen in einem systematischen Review, dass LLMs zunehmend in der Praxis eingesetzt werden, z. B. um User Stories auf Konsistenz und Vollständigkeit zu prüfen oder Vorschläge für präzisere Formulierungen zu generieren. [13] Studien wie von Ferrari et al. verdeutlichen, dass insbesondere Mehrdeutigkeit und implizites Wissen („tacit knowledge“) durch KI-gestützte Analysen reduziert werden können. [17]

Gleichzeitig weisen neuere Arbeiten auf Herausforderungen hin: KI-gestützte Systeme sind stark abhängig von Prompt-Design, Datenqualität und Modelltransparenz. Montgomery et al. betonen, dass trotz des Potenzials für Qualitätssteigerung auch Risiken bestehen – etwa die Einführung neuer Verzerrungen (Bias) oder die Reduktion komplexer Anforderungen auf rein syntaktische Prüfungen. [30]

## LLM-Grundlagen

Um die in Kapitel 2.2 beschriebenen Entwicklungen im Requirements Engineering im Zusammenspiel mit KI besser zu verstehen, ist es notwendig, die technologischen Grundlagen der Large Language Models (LLMs) darzustellen. Kapitel 2.3 führt daher zunächst in die Funktionsweise, anschließend in die Interaktionsmethoden (Prompt-Engineering) sowie in das Konzept des Function-Calling ein, das die Anbindung externer Werkzeuge ermöglicht.

### Funktionsweise

A diagram of a process

AI-generated content may be incorrect.Large Language Models basieren auf Transformer-Architekturen, die erstmals von Vaswani et al. vorgestellt wurden. Das zentrale Prinzip dieser Architektur ist der Self-Attention-Mechanismus, der es ermöglicht, Abhängigkeiten zwischen weit auseinanderliegenden Token in einer Eingabesequenz effizient zu modellieren. Dadurch können LLMs kontextabhängige Bedeutungen erfassen und kohärente Texte generieren. [1]

Abbildung Transofrmoer Architektur

Die Leistungsfähigkeit moderner LLMs ergibt sich vor allem aus ihrer Skalierung. Studien zeigen in einer umfassenden Übersicht, dass Modellgröße (Anzahl der Parameter), Datenqualität und Trainingsdauer [11] in direkter Korrelation mit der erzielten Performance stehen. Hou et al. betonen, dass LLMs über rein syntaktische Muster hinaus auch semantische Strukturen erfassen können und dadurch in der Lage sind, komplexe Aufgaben wie Codegenerierung, Bugfixing und Refactoring im Software Engineering zu bewältigen. [4]

Neben dem Pretraining auf großen, heterogenen Text- und Codekorpora kommt zunehmend „Instruction Tuning“ und „Reinforcement Learning from Human Feedback (RLHF)“ zum Einsatz. Diese Verfahren passen LLMs an spezifische Aufgaben an und steigern ihre Nützlichkeit in praxisnahen Szenarien. [2]

### Prompt-Engineering

A diagram of a system

AI-generated content may be incorrect.Während klassische Software durch explizite Programmierung gesteuert wird, interagiert man mit LLMs primär über Prompts, also natürlichsprachige Eingaben. Die Gestaltung dieser Prompts bekannt als Prompt-Engineering hat entscheidenden Einfluss auf die Modellleistung. [6] White et al. beschreiben Prompt-Engineering als eigenständige Disziplin, die über das klassische Few-Shot-Lernen hinausgeht. Zu den etablierten Techniken zählen Chain-of-Thought (CoT), Role-Prompting und Self-Consistency. [33]

Abbildung Chain of Thought concepts

Empirische Studien verdeutlichen jedoch die Fragilität von Prompts. Zamfirescu-Pereira et al. zeigen, dass bereits geringfügige Änderungen in der Formulierung zu stark abweichenden Ergebnissen führen können. [34] Diese Sensitivität unterstreicht die Notwendigkeit einer systematischen Herangehensweise. In der Praxis entstehen daher zunehmend Prompt-Bibliotheken und Prompt-as-a-Service-Plattformen, die wiederverwendbare, getestete Prompts bereitstellen. [6]

Für den Einsatz im Software Engineering bedeutet dies, dass durch gezielte Prompt-Gestaltung spezifische Aufgaben wie Anforderungsanalyse, Codegenerierung oder Testfallentwurf effektiv unterstützt werden können ein Aspekt, der insbesondere im Kontext von GitLab-Workflows (Kapitel 4) relevant wird.

### Function-Calling

Eine Weiterentwicklung moderner LLMs besteht in der Fähigkeit zum Function-Calling, also der strukturierten Generierung von Funktionsaufrufen, die externe APIs oder Tools ansteuern. Anstatt nur Text auszugeben, kann ein Modell damit gezielt Datenbankabfragen formulieren, Code ausführen oder Softwarewerkzeuge steuern. [35]

Dieses Konzept ermöglicht es, LLMs als Orchestratoren hybrider Systeme einzusetzen: Sie verbinden natürlichsprachliche Steuerung mit der präzisen Ausführung externer Funktionen. [7] Im Kontext von Software-Engineering-Workflows bedeutet dies, dass LLMs beispielsweise GitLab-APIs ansprechen, Pipelines starten oder Merge Requests automatisiert kommentieren können.

A diagram of a software process

AI-generated content may be incorrect.

Abbildung function Calling

Allerdings sind mit Function-Calling auch Sicherheitsrisiken verbunden. Es weisen verschiedene Studien auf Angriffsvektoren wie Prompt Injections und Protokoll-Exploits hin, die entstehen können, wenn LLMs ohne ausreichende Validierung externe Schnittstellen ansteuern. Für den Einsatz in kritischen Entwicklungsumgebungen ist daher ein besonderes Augenmerk auf Transparenz, Zugriffskontrolle und Auditing erforderlich. [36]

## Agentsysteme

Agentensysteme sind ein zentrales Konzept in der Informatik und Künstlichen Intelligenz (KI). Sie dienen dazu, komplexe Probleme durch autonome, interagierende Softwareeinheiten zu lösen. Während klassische Agentenmodelle auf logischen und regelbasierten Paradigmen aufbauen, erweitern neuere Entwicklungen – insbesondere durch den Einsatz von Large Language Models (LLMs) – das Spektrum um datengetriebene und probabilistische Ansätze. Dieses Kapitel führt zunächst in die Grundlagen ein (2.4.1), erläutert klassische Agentensysteme (2.4.2) und stellt anschließend die modernen Entwicklungen im Kontext von LLMs dar (2.4.3).

### Grundlagen von Agentensystemen

Ein Agent wird in der Literatur als eine **autonome Softwareeinheit** beschrieben, die in der Lage ist, ihre Umgebung wahrzunehmen, darauf zu reagieren und eigene Handlungen auszuführen, um bestimmte Ziele zu verfolgen. [37] Jennings, Sycara und Wooldridge definieren vier wesentliche Eigenschaften von Agenten [38]:

1. **Autonomie** – sie handeln unabhängig und kontrollieren ihren eigenen Zustand.
2. **Reaktivität** – sie reagieren auf Veränderungen in der Umgebung in Echtzeit.
3. **Proaktivität** – sie verfolgen eigene Ziele und planen Handlungen.
4. **Sozialität** – sie interagieren mit anderen Agenten oder Menschen.

Diese Merkmale unterscheiden Agenten von klassischen Softwaresystemen, die in der Regel deterministisch und zentralisiert arbeiten. Russell und Norvig betonen, dass Agenten als „Rational Agents“ verstanden werden können, die versuchen, ihre Leistung anhand einer definierten Nutzenfunktion zu maximieren. [39]

Ein **Multi-Agent-System (MAS)** besteht aus mehreren dieser autonomen Einheiten, die durch Kooperation, Koordination oder Wettbewerb zusammenarbeiten. MAS werden seit den 1990er Jahren in Bereichen wie **Robotik, Simulation, verteilte Systeme und Logistik** eingesetzt. [37, 38]

### Klassische Agentensysteme

Die klassischen Ansätze zu Agentensystemen wurden maßgeblich durch logische und symbolische Methoden geprägt. Ein prominentes Modell ist das BDI-Modell (Belief, Desire, Intention), das Agenten durch ihre Wissensbasis (Beliefs), ihre Ziele (Desires) und ihre geplanten Handlungen (Intentions) beschreibt. [37, 40] Dieses Modell diente als theoretische Grundlage für zahlreiche Implementierungen von Agentensystemen in den 1990er und 2000er Jahren.

A diagram of a flowchart

AI-generated content may be incorrect.

Abbildung BDI Aktivitätsdiagramm [40]

Ein weiteres etabliertes Paradigma sind Blackboard-Systeme, bei denen mehrere Agenten über eine gemeinsame Datenstruktur interagieren. Dabei dient das „Blackboard“ als Koordinationsfläche, auf der Agenten Wissen austauschen und ihre Handlungen synchronisieren.

Zur Kommunikation in MAS wurden Standards wie die Agent Communication Language (ACL) und die FIPA-Standards entwickelt. Diese ermöglichen es, Nachrichten zwischen Agenten in strukturierter Form auszutauschen und komplexe Kooperationsprotokolle abzubilden. [38, 41]

Calegari et al. zeigen in einer systematischen Literaturübersicht, dass klassische MAS insbesondere auf formalen Logiksystemen aufbauen, um Eigenschaften wie Korrektheit, Vorhersagbarkeit und Stabilität sicherzustellen. Gleichzeitig weisen sie darauf hin, dass diese Systeme häufig in hochspezialisierten Anwendungsdomänen zum Einsatz kamen (z. B. Fertigung, E-Commerce), jedoch Schwierigkeiten bei der Skalierung und Generalisierung auftraten. [42]

Damit bilden klassische MAS eine solide theoretische Basis, deren Stärken in Formalisierung und Nachvollziehbarkeit liegen, deren Limitationen jedoch in der eingeschränkten Flexibilität und der geringen Adaptivität an neue, offene Umgebungen deutlich werden.

### Agenten im Kontext von LLMs

Mit der Einführung von Large Language Models hat sich das Agentenkonzept grundlegend erweitert. LLMs ermöglichen es, Agenten zu konstruieren, die nicht mehr ausschließlich auf logischen Regeln und vordefinierten Protokollen beruhen, sondern auf statistischen Sprachmodellen, die in der Lage sind, komplexe Aufgaben flexibel zu bearbeiten. [4]

Guo et al. beschreiben LLM-basierte Agenten als Systeme, die Planung, Gedächtnis, Tool-Zugriff und Kommunikation integrieren können. Diese Eigenschaften resultieren nicht aus explizit programmierten Regeln, sondern aus emergenten Fähigkeiten, die durch das Training auf großen Datenmengen entstehen. [43] Plaat et al. bezeichnen diese Systeme als „agentic LLMs“ und argumentieren, dass sie im Gegensatz zu klassischen MAS nicht nur symbolische Repräsentationen verarbeiten, sondern direkt mit unstrukturierten Daten interagieren können. [44]

Im Software Engineering werden LLM-Agenten zunehmend in mehragentigen Workflows eingesetzt, etwa für Planung, Codierung, Testen und Review. Wang et al. zeigen in ihrer systematischen Übersicht, dass diese Systeme neue Formen der Zusammenarbeit ermöglichen: LLM-Agenten kommunizieren nicht über formale ACL-Protokolle, sondern über natürliche Sprache, und können dabei externe Werkzeuge über APIs oder Function-Calling ansteuern. [8]

Applis et al. skizzieren die Vision eines „Unified Software Engineering Agent“, der den gesamten Entwicklungszyklus von der Anforderungsanalyse über Codierung und Testing bis zur Integration weitgehend autonom bewältigen kann. [8] Gleichzeitig weist Botti kritisch darauf hin, dass viele der diskutierten Eigenschaften von LLM-Agenten bereits in der klassischen Agentenforschung untersucht wurden. Die Frage, ob LLM-Agenten tatsächlich eine völlig neue Paradigmenklasse darstellen oder lediglich bestehende Konzepte erweitern, ist daher eine offene Forschungsdiskussion. [45]

LLM-Agenten ABBILDUNG

### Vergleich: Klassische vs. LLM-basierte Agenten

Während klassische Agentensysteme durch **formale, logische Modelle** geprägt sind, basieren LLM-Agenten auf **datengetriebenen, probabilistischen Modellen**. Beide Ansätze teilen zentrale Eigenschaften wie **Autonomie, Reaktivität und Sozialität**, unterscheiden sich jedoch deutlich in ihrer Architektur, Kommunikation und Lernfähigkeit. Tabelle 1 fasst die zentralen Unterschiede zusammen.

| **Aspekt** | **Klassische Agenten** | **LLM-basierte Agenten** |
| --- | --- | --- |
| **Architektur** | Logik- und regelbasiert (z. B. BDI-Modell)[37, 38] | Transformer-basierte Sprachmodelle, datengetrieben[4, 43] |
| **Wissensrepräsentation** | Explizit, symbolisch (Beliefs, Regeln, Wissensbasen) [37, 42] | Implizit, verteilt in hochdimensionalen Vektorräumen[43] |
| **Kommunikation** | Standardisierte Protokolle (ACL, FIPA)[38, 41] | Natürliche Sprache, API-Aufrufe, Function-Calling[8, 43] |
| **Entscheidungsfindung** | Deterministisch, logische Ableitung, Planer [37, 42] | Probabilistisch, emergente Fähigkeiten (Planung, Tool-Use)[43, 44] |
| **Lernfähigkeit** | Gering, erfordert manuelles Nachrüsten/Programmieren [37, 39] | Hoch, durch Pretraining, Fine-Tuning und Few-Shot-Lernen[4, 43] |
| **Adaptivität** | Eingeschränkt, nur für definierte Umgebungen geeignet [42] | Hoch, flexibel einsetzbar in offenen Umgebungen[8, 44] |
| **Transparenz** | Hoch, da logische Modelle nachvollziehbar sind[37, 42] | Niedrig, „Black-Box“-Charakter erschwert Erklärbarkeit[4, 45] |
| **Beispiele** | E-Commerce-Agenten, Fertigungssteuerung, Simulationen[38, 42] | AutoGPT, LangChain, MetaGPT, GitLab-Agents[7, 8, 44] |

Tabelle Agenten Vergleich

Klassische Agentensysteme sind vor allem durch ihre Formalität und Transparenz gekennzeichnet. Sie bieten klare Modelle für Wissen, Kommunikation und Planung (z. B. BDI), eignen sich jedoch vorwiegend für statisch definierte Szenarien [1], [2]. Ihre Stärke liegt in Nachvollziehbarkeit und Korrektheit, ihre Schwäche in mangelnder Flexibilität.

LLM-basierte Agenten hingegen zeichnen sich durch emergente Intelligenz aus. Anstatt formale Regeln zu befolgen, generieren sie Handlungen probabilistisch auf Grundlage großer Trainingsdaten. Dadurch sind sie in der Lage, neue Situationen flexibel zu bewältigen, externe Werkzeuge anzusteuern und sogar natürlichsprachliche Kommunikation als universelles Interface zu nutzen [7], [9]. Gleichzeitig bringen sie Herausforderungen mit sich: ihre Entscheidungen sind schwer erklärbar, und ihre Ergebnisse variieren stark je nach Prompt, Kontext und Datenqualität [6], [11].

Damit zeigt sich ein Spannungsfeld: Klassische Agenten bieten Struktur und Sicherheit, während LLM-Agenten Flexibilität und Autonomie in dynamischen Umgebungen ermöglichen. In der Praxis könnten hybride Ansätze logikbasierte Steuerung kombiniert mit LLM-basierten Modulen – zukünftig eine Brücke zwischen beiden Welten schlagen.

### Supervisor- und Orchestrator-Pattern

Die Koordination mehrerer Agenten ist eine der zentralen Herausforderungen in Multi-Agent-Systemen (MAS). Ohne geeignete Steuerungsmechanismen können Konflikte entstehen, redundante Berechnungen ausgeführt oder Ressourcen ineffizient genutzt werden. In der Literatur haben sich verschiedene Orchestrierungsansätze etabliert, die sich im Grad der Zentralisierung und in der Form der Interaktion unterscheiden. [37]

**Supervisor-Pattern**

Das Supervisor-Pattern beschreibt ein Koordinationsmodell, in dem ein zentraler Agent die Steuerung übernimmt. Der Supervisor verteilt Aufgaben an spezialisierte Sub-Agenten, überwacht deren Ausführung und aggregiert die Ergebnisse. [37] Dadurch entsteht eine hierarchische Kontrollstruktur, die eine hohe Nachvollziehbarkeit und klare Verantwortlichkeiten bietet.

* **Vorteile:** Transparenz, deterministische Abläufe, vereinfachte Fehlerbehandlung.
* **Nachteile:** Abhängigkeit von einem zentralen Knoten (Single Point of Failure), eingeschränkte Flexibilität in dynamischen Umgebungen.

Dieses Muster ist insbesondere in hierarchischen Organisationen und in Szenarien verbreitet, in denen **Reproduzierbarkeit und Kontrolle** im Vordergrund stehen. [8, 37]

**Orchestrator-Pattern**

Ein allgemeineres Modell ist das **Orchestrator-Pattern**, das verschiedene Formen der Koordination umfasst. Hierbei kann ein zentraler Orchestrator Agenten **sequentiell**, **parallel** oder in **Feedback-Schleifen** organisieren. [46]

* **Sequentielle Orchestrierung:** Agenten arbeiten nacheinander in einer festen Reihenfolge (z. B. Planung → Ausführung → Evaluation).
* **Parallele Orchestrierung:** Mehrere Agenten arbeiten gleichzeitig an Teilaufgaben, deren Ergebnisse später zusammengeführt werden.
* **Iterative Orchestrierung:** Ergebnisse einzelner Agenten werden wiederholt zwischen ihnen ausgetauscht, bis ein stabiler Zustand erreicht ist.

Die Literatur unterscheidet hier zwischen **zentralisierter Orchestrierung** (ein Orchestrator-Agent koordiniert alle anderen) und **dezentraler Koordination** (Agenten handeln autonom und stimmen sich über Kommunikationsprotokolle ab). [8, 38] Während zentrale Ansätze durch Einfachheit und Kontrolle überzeugen, sind dezentrale Systeme robuster gegenüber Ausfällen und besser skalierbar, erfordern jedoch komplexere Kommunikationsmechanismen.

### Herausforderungen

Die Koordination einer großen Anzahl von Agenten erfordert erhebliche Ressourcen und ausgefeilte Mechanismen. Klassische MAS-Architekturen stoßen bei wachsender Agentenzahl oft auf **Kommunikations- und Synchronisationsprobleme**. [37] Bei LLM-basierten Agenten verschärft sich diese Problematik durch den hohen Rechenaufwand und die Abhängigkeit von leistungsstarker Hardware. Guo et al. betonen, dass insbesondere die Orchestrierung vieler LLM-Agenten noch ungelöst ist und häufig zu ineffizientem Ressourcenverbrauch führt. [43]

Ein zentrales Problem besteht in der **Fehlerpropagation**: Fehler einzelner Agenten können sich im System verstärken und zu unvorhersehbaren Ergebnissen führen. [37] Bei LLM-Agenten kommt hinzu, dass ihre Antworten probabilistisch sind und somit eine inhärente Varianz aufweisen. Plaat et al. weisen darauf hin, dass diese Stochastik die Reproduzierbarkeit von Experimenten erheblich einschränkt ein Aspekt, der für wissenschaftliche Evaluationen besonders kritisch ist. [44]

Agentensysteme sind anfällig für **Manipulationen und Angriffe**. In klassischen MAS betrifft dies vor allem Kommunikationsprotokolle. Bei LLM-Agenten treten zusätzliche Risiken auf, wie **Prompt Injections** oder **unbeabsichtigte API-Aufrufe**. Botti warnt, dass LLM-basierte Agenten durch ihre Offenheit für unstrukturierte Eingaben besonders verwundbar sind und neue Angriffsflächen eröffnen, die in der klassischen Agentenforschung kaum berücksichtigt wurden. [45]

Die Bewertung von Agentensystemen bleibt eine offene Herausforderung. Während klassische MAS durch ihre logikbasierte Struktur relativ gut analysierbar sind, erschwert der „Black-Box“-Charakter von LLMs die **Erklärbarkeit und Nachvollziehbarkeit** ihrer Entscheidungen (Kapitel 2.3) hat gezeigt, dass es bislang an einheitlichen Benchmarks und Metriken mangelt, um Prozess- und Ergebnisqualität zuverlässig zu messen.

## Model Context Protocol

A diagram of a server

AI-generated content may be incorrect.Das Model Context Protocol (MCP) ist ein offenes Protokoll zur standardisierten Anbindung externer Datenquellen und Werkzeuge an LLM-Anwendungen. Es zielt darauf ab, Kontext und Fähigkeiten portabel verfügbar zu machen—oft verkürzt als „USB-C für AI-Apps“ beschrieben. Formal definiert die Spezifikation die verbindlichen Protokollanforderungen; Begleitdokumente erläutern Einsatzmuster und Entwickler-Guides. [15]

Abbildung MCP - Architektur

MCP folgt einem Client-Server-Modell mit Fähigkeiten-Aushandlung beim Verbindungsaufbau: Der Client initiiert mit initialize, der Server antwortet mit unterstützten Features danach stehen Server-Fähigkeiten bereit. Der Standard unterscheidet insbesondere

1. **Tools** (aufrufbare Aktionen/Funktionen),
2. **Resources** (kontextuelle Inhalte über URI-artige Bezüge) und
3. **Prompts** (strukturierte Prompt-Vorlagen mit Parametrisierung).

Zusätzlich sieht die Spezifikation **Utilities** (z. B. Telemetrie, Protokollierung) vor. Das Design ist **modular**, d. h. Implementierungen **müssen** den Basisteil inkl. Lebenszyklus unterstützen und **können** weitere Komponenten je nach Bedarf ergänzen.

MCP kodiert Nachrichten über **JSON-RPC 2.0** (UTF-8). Standardisierte **Transporte** sind derzeit **stdio** (Standard-Ein/Ausgabe) und **Streamable HTTP**; eigene Transporte sind explizit erweiterbar. In der Community wird zudem ein **WebSocket-Transport** diskutiert, ist aber (Stand: 2025) kein Teil der Kern-Spezifikation.

Untersuchungen heben hervor, dass MCP wie jede Tool-Schnittstelle **Angriffsflächen** eröffnet (u. a. **Tool-Poisoning**, indirekte Prompt-Injection, Kettenangriffe über geteilten Kontext). Aktuelle Analysen schlagen **Taxonomien** und **Angriffsbibliotheken** vor und betonen die Notwendigkeit robuster **Kontrollen**.

# Forschungsdesign & Methodik

Dieses Kapitel legt das Forschungsdesign fest, mit dem die in Kapitel 1 formulierten Ziele und Forschungsfragen beantwortet werden. Methodisch ist die Arbeit im Design-Science-Paradigma (DSR) verankert: Ein Artefakt wird entworfen und unter realitätsnahen Bedingungen evaluiert, um wissenschaftliche Strenge und praktische Relevanz zu verbinden. [18, p. 46] Zugleich adressiert das Design die Besonderheiten LLM-basierter Agentensysteme, bei denen Reproduzierbarkeit, Robustheit und mehrdimensionale Metriken zentrale Bewertungsachsen darstellen. [11, p. 1-3]

## Design-Science-Research

A diagram of a company

AI-generated content may be incorrect.DSR ist ein in den Informations- und Softwarewissenschaften etablierter Ansatz zur Entwicklung, Demonstration und Evaluation neuartiger Artefakte (Konstrukte, Modelle, Methoden, Instanzen) mit praktischer Relevanz und wissenschaftlichem Erkenntnisgewinn. Die Stärke des Ansatzes liegt in der engen Kopplung von Artefaktentwicklung, theoretischer Einbettung und systematischer Evaluation. [18, p. 46]

Das weithin rezipierte DSR-Prozessmodell von Peffers et al. umfasst sechs Schritte und dient als Referenzrahmen für die vorliegende Arbeit:

Abbildung DSRM Prozess Modell [18, p. 54-55]

Studien zur Evaluation von Large Language Models betonen, dass komplexe Systeme mehrdimensionale Evaluationskriterien (z. B. Ergebnis-, Prozess- und Robustheitsmetriken) benötigen und dass Reproduzierbarkeit explizit adressiert werden muss. [11, pp. 1–3, 25–27] Entsprechend berücksichtigt diese Arbeit neben funktionalen Ergebnissen (z. B. Merge-Erfolg) Prozessmetriken (z. B. Durchlaufzeiten, Fehlerraten), um Leistungsfähigkeit realitätsnah zu erfassen.

Für LLM-Agentensysteme ist eine transparente Dokumentation von Parametern (Prompts, Seeds, Modelleinstellungen) und eine kontrollierte Versuchsanordnung zentral, um robuste und nachvollziehbare Aussagen zu ermöglichen. [11]

## Untersuchungsmodell & Hypothesen

Das Untersuchungsmodell dieser Arbeit orientiert sich an den in Kapitel 1 skizzierten Forschungsfragen (FF.1 und FF.2) und integriert die theoretischen Grundlagen aus Kapitel 2. Ziel ist es, die Wirkung der syntaktischen Anforderungsqualität auf die Performanz von LLM-Agenten in GitLab-Workflows sowie die Robustheit der Ergebnisse bei wiederholten Durchläufen systematisch zu erfassen.

### Untersuchungsmodell

Die Struktur folgt einer DSR-basierten Evaluationsstudie mit unabhängigen Variablen (UV), abhängigen Variablen (AV) und Kontrollvariablen (KV) [18, p. 46]:

**UV – Qualität der Anforderungen.**

* hoch: Formulierung gemäß ISO/IEC/IEEE 29148:2018; Qualitätsmerkmale für einzelne Anforderungen umfassen u. a. necessary, unambiguous, complete, singular, feasible, verifiable, correct, conforming. [12, pp. 19–20]
* niedrig: bewusst mehrdeutige, unpräzise und unstrukturierte Texte, die typische Ambiguitäten natürlicher Sprache aufweisen; Ambiguität wirkt sich nachweislich störend auf die Elicitation und Dokumentation aus, kann aber auch latentes Wissen sichtbar machen. [17, pp. 1–3]

**AV – Erfolgs- und Prozessmetriken.**

* Erfolgsmetriken: MR-Erfolg, Pipeline-Erfolg, Korrektheit der Bugfixes.
* Prozessmetriken: Durchlaufzeit, Fehlerrate, Anzahl Review-Zyklen.

Die Wahl mehrerer, voneinander unterschiedlicher Metriken folgt Empfehlungen für LLM-Evaluation (what/where/how to evaluate) und Robustheitsbetrachtungen.

**KV – Konstante Rahmenbedingungen.**

* 1. verwendetes LLM-Modell, Provider und Temperatur,
  2. Prompts/Scaffold-Strukturen,
  3. identisches GitLab-Setup,
  4. Schnittstellenkonfiguration. LLMs reagieren sensitiv auf Prompt-Variationen; konsistente Prompt- und Tool-Setups reduzieren Störvariabilität. [11, p. 1]

**Störfaktor – Stochastik.**

LLM-Generierung beinhaltet probabilistische Komponenten: Temperierung/Sampling und Seed-Setzung beeinflussen Variation. Reproduzierbarkeitsfragen werden in der LLM-Evaluationsliteratur als zentral herausgestellt ([11, pp. 25–27]).

### Hypothesen

Vor diesem Hintergrund ergeben sich zwei Hypothesen, die eng mit den Forschungsfragen verknüpft sind:

**Hypothese 1 zu Forschungsfrage 1:**

Eine höhere syntaktische Qualität von Anforderungen gemäß ISO/IEC/IEEE 29148:2018 und QUS-Kriterien führt zu signifikant besseren Erfolgsmetriken (MR-/Pipeline-Erfolg, Korrektheit) und Prozessmetriken (Durchlaufzeit, Fehlerrate) in GitLab-basierten Simulationen.

Begründung: Empirische Studien zeigen, dass klare, eindeutige Anforderungen den Projekterfolg maßgeblich beeinflussen [12, 29], während Mehrdeutigkeit und unklare Formulierungen zu Implementierungsfehlern führen. [17, 30]

**Hypothese 2 zu Forschungsfrage 2:**

Wiederholte Simulationen mit identischen Parametern und Konfigurationen zeigen eine messbare Varianz in Erfolgs- und Prozessmetriken, diese bleibt jedoch innerhalb eines stabilen Korridors, der eine kontrollierte Reproduzierbarkeit gewährleistet.

Begründung: LLM-Agenten agieren probabilistisch und reagieren sensibel auf minimale Prompt- oder Konfigurationsänderungen. [16] Gleichzeitig ermöglichen kontrollierte Rahmenbedingungen (fixierte Seeds, standardisierte Pipelines, MCP) die Eingrenzung der Varianz auf ein reproduzierbares Niveau. [11, 14]

## Versuchsplanung

Die Evaluation erfolgt in Form kontrollierter Simulationen auf Basis von GitLab-Workflows. Ziel ist es, die beiden in Kapitel 1.3 formulierten Forschungsfragen empirisch zu beantworten. Dazu werden zwei Projektszenarien aufgesetzt, die sich ausschließlich in der syntaktischen Qualität der Anforderungen unterscheiden, während alle weiteren Rahmenbedingungen konstant gehalten werden.

**Experimentelles Setup**

Die Simulationsumgebung orientiert sich an etablierten Design-Science-Prinzipien und wird so gestaltet, dass die Vergleichbarkeit zwischen den Szenarien gewährleistet ist.

* **Projekt A (hohe Anforderungsqualität):** Enthält Issues, die nach ISO/IEC/IEEE 29148 eindeutig, präzise und testbar formuliert sind.
* **Projekt B (niedrige Anforderungsqualität):** Enthält identische Issues mit denselben funktionalen Inhalten, jedoch in sprachlich verschachtelter, mehrdeutiger und unstrukturierter Form, die typische Probleme unklarer Anforderungen widerspiegeln.

Beide Projekte sind als GitLab-Templates implementiert und enthalten standardisierte CI/CD-Pipelines (Build, Test), die es ermöglichen, Erfolgs- und Prozessmetriken reproduzierbar zu erfassen.

**Agentenbasierte Workflow-Orchestrierung**

Die Bearbeitung der Issues erfolgt durch ein mehragentiges System, das auf den in Kapitel 2.4 beschriebenen Inhalten basiert.

Die Orchestrierung stellt sicher, dass die Unterschiede zwischen Projekt A und B ausschließlich auf die Formulierung der Anforderungen zurückgeführt werden können.

Um die interne Validität des Experiments sicherzustellen, werden alle externen Einflussfaktoren kontrolliert:

* identische GitLab-Umgebungen (Runner, Konfigurationen, Pipeline-Definitionen),
* identisches Repository-Setup, identische Codebasis,
* gleiche LLM-Modelle und Provider, fixierte Konfigurationen (Seeds, Temperatur),
* standardisierte Prompts für die Agenten.

Diese Maßnahmen reduzieren die Einflüsse stochastischer Varianz, sodass Unterschiede in den Ergebnissen primär durch die syntaktische Qualität der Anforderungen erklärt werden können.

## Messkonzept & Instrumentierung

Zur Beantwortung der Forschungsfragen werden Erfolgs- und Prozessmetriken erhoben, die sowohl die Effektivität der Agenten (FF.1) als auch die Stabilität ihrer Ergebnisse bei wiederholten Runs (FF.2) abbilden. Die Messung erfolgt über GitLabs eigenes Überwachungssystem.

### Erfolgsmetriken

Diese Metriken geben Auskunft über die Zielerreichung im GitLab-Workflow:

* **Merge-Request-Erfolg (%):** Anteil der erfolgreichen MRs an allen Versuchen.
* **Pipeline-Erfolg (%):** Anteil fehlerfrei durchgelaufener CI/CD-Pipelines.
* **Bugfix-Korrektheit (%):** Anteil korrekt gelöster Issues (manuelle Validierung anhand Testfällen).

### Prozessmetriken

Diese Metriken beschreiben die Effizienz und Stabilität der Abläufe:

* **Durchlaufzeit (min):** Zeitspanne von Issue-Start bis erfolgreichem Merge.
* **Fehlerrate (%):** Anteil gescheiterter Builds oder fehlgeschlagener Tests.
* **Review-Zyklen (Anzahl):** Wie viele Agenten-Iteration(en) notwendig waren, bis ein MR akzeptiert wurde.

### Berechnungen

Es werden Mittelwerte, Erfolgsquoten und Streuungsmaße (σ, Varianz) berechnet, um die Forschungsfragen zu beantworten:

* **Mittelwert (𝑥̄):** Durchschnittlicher Wert pro Metrik über alle Runs.
* **Standardabweichung (σ):** Maß für die Streuung der Ergebnisse → direkt relevant für FF.2.
* **Erfolgsquoten (%):** Anteil erfolgreicher Runs (z. B. MRs, Pipelines).
* **Varianzvergleich:** Gegenüberstellung von Projekt A (gute Anforderungen) und Projekt B (schlechte Anforderungen), um Hypothese H1 und H2 zu prüfen.

Für die Ergebnisdarstellung werden Mittelwerte und Streuungen tabellarisch und grafisch (Balken-/Boxplots) visualisiert. Dadurch lassen sich Unterschiede zwischen den Projekten sowie die Stabilität der Runs leicht nachvollziehen.

Damit ist gewährleistet, dass sowohl Effektivität als auch Reproduzierbarkeit in einem einfachen, aber aussagekräftigen Messkonzept berücksichtigt werden.

## Auswertungsverfahren

Die in Kapitel 3.4 beschriebenen Messwerte werden nach Abschluss der Simulationen systematisch aufbereitet und analysiert. Ziel der Auswertung ist es, Unterschiede in der Effektivität der Agenten zwischen Szenarien mit hoher und niedriger Anforderungsqualität (FF.1) sowie die Stabilität der Ergebnisse bei wiederholten Durchläufen unter identischen Bedingungen (FF.2) transparent darzustellen.

**Datenaufbereitung**:

Alle im Rahmen der Simulation erhobenen Metriken werden anhand von GitLab ermittelt. Jede einzelne Simulation (Run) stellt dabei eine Beobachtungseinheit dar, sodass eine konsistente Vergleichsbasis zwischen Projekt A und Projekt B gewährleistet ist. Zusätzlich werden Metadaten wie Zeitstempel, verwendetes Modell, Konfigurationen sowie Random-Seeds dokumentiert, um Reproduzierbarkeit und Nachvollziehbarkeit der Ergebnisse sicherzustellen.

**Deskriptive Statistik:**

Die Auswertung erfolgt primär deskriptiv, um die Ergebnisse klar und nachvollziehbar zu halten. Für jede Metrik werden Mittelwerte (𝑥̄), Medianwerte sowie Standardabweichungen (σ) berechnet. Erfolgsquoten (z. B. Anteil erfolgreicher Merge Requests oder fehlerfreier Pipelines) werden in Prozent angegeben. Zur Veranschaulichung werden die Ergebnisse tabellarisch zusammengefasst und zusätzlich visuell dargestellt – Mittelwerte über Balkendiagramme, Streuungen und Ausreißer über Boxplots. Auf diese Weise lassen sich Unterschiede und Muster zwischen den Projektszenarien anschaulich herausarbeiten.

**Vergleich zwischen Projekt A und Projekt B (FF.1):**

Um die Wirkung der Anforderungsqualität auf die Performanz der Agenten zu prüfen, werden die Metriken beider Projekte gegenübergestellt. Im Mittelpunkt steht die Frage, ob sich konsistente Unterschiede in Erfolgs- und Prozessmetriken identifizieren lassen. Neben der direkten Gegenüberstellung von Mittelwerten und Erfolgsquoten werden einfache Effektgrößen (z. B. Differenz der Mittelwerte Δ𝑥̄) berechnet, um die praktische Relevanz der Unterschiede zu bewerten.

**Analyse der Varianz und Stabilität (FF.2):**

Zur Beantwortung der zweiten Forschungsfrage wird die Streuung innerhalb wiederholter Runs analysiert. Standardabweichung und Varianz geben Aufschluss darüber, in welchem Ausmaß die Ergebnisse trotz identischer Rahmenbedingungen schwanken. Die Bewertung konzentriert sich darauf, ob die beobachtete Varianz innerhalb eines stabilen Korridors bleibt und damit eine hinreichende Reproduzierbarkeit gewährleistet ist.

**Interpretation:**

Die quantitativen Ergebnisse werden abschließend mit den Hypothesen H1 und H2 interpretiert und den theoretischen Grundlagen aus Kapitel 2 verknüpft. Dabei wird sowohl die praktische Bedeutung (z. B. Einfluss klarer Anforderungen auf Pipeline-Erfolg) als auch die methodische Relevanz (z. B. Rolle der stochastischen Varianz in LLM-basierten Workflows) reflektiert. Damit ist sichergestellt, dass die Auswertung nicht nur statistische Unterschiede beschreibt, sondern diese auch inhaltlich in Bezug auf die Forschungsfragen einordnet.

# Systementwurf

Dieses Kapitel beschreibt den konzeptionellen Entwurf des Artefakts, das zur Beantwortung der Forschungsfragen entwickelt wurde. Der Entwurf folgt dem in Kapitel 3.1 dargelegten Design-Science-Research-Ansatz und setzt die in Kapitel 2 erarbeiteten theoretischen Grundlagen in eine praktische Architektur um. Im Mittelpunkt steht die Konzeption einer Simulationsumgebung, die vollständige Software-Engineering-Workflows in GitLab Ende-zu-Ende orchestriert und dabei sowohl die Wirkung der Anforderungsqualität auf die Agenten-Performanz als auch die Reproduzierbarkeit der Ergebnisse systematisch messbar macht.

## Anforderungen

Die Anforderungen an das zu entwickelnde System leiten sich aus dem Forschungsziel ab, einen Machbarkeitsnachweis für die vollständige End-to-End-Orchestrierung von Software-Engineering-Workflows durch Agentensysteme zu erbringen. Der Fokus liegt dabei auf der Demonstration der grundsätzlichen Funktionsfähigkeit eines solchen Systems und der Schaffung einer Evaluationsbasis für die formulierten Forschungsfragen. Als Proof-of-Concept ist das System bewusst auf die Kernfunktionalitäten beschränkt, die für den wissenschaftlichen Nachweis notwendig sind.

### Funktionale Anforderungen

**Projektanalyse und Planung:** Das System muss in der Lage sein, ein GitLab-Projekt vollständig zu analysieren. Dies umfasst das Lesen der Repository-Struktur, das Identifizieren der verwendeten Programmiersprachen und Frameworks sowie das Extrahieren offener Issues mit allen relevanten Metadaten wie Titel, Beschreibung, Labels und Prioritäten. Auf Basis dieser Analyse muss das System einen strukturierten Implementierungsplan erstellen, der die Issues nach Priorität und Abhängigkeiten ordnet und für jedes Issue einen groben Lösungsansatz skizziert.

**Code-Implementierung:** Das System muss für jedes zu bearbeitende Issue selbstständig einen Feature-Branch erstellen und die notwendigen Code-Änderungen vornehmen können. Dies beinhaltet das Erstellen neuer Dateien, das Modifizieren bestehender Dateien und das Löschen obsoleter Komponenten. Die Implementierung muss den technischen Kontext des Projekts berücksichtigen, das heißt die verwendeten Frameworks, Architekturmuster und Coding-Konventionen respektieren. Nach Abschluss der Implementierung muss das System die Änderungen in den Feature-Branch committen und nach GitLab pushen können. Diese Anforderung bildet den Kern der E2E-Orchestrierung und ist zentral für die Evaluation, da hier der Einfluss der Anforderungsqualität auf die Code-Generierung untersucht wird.

**Test-Generierung und Qualitätssicherung:** Das System muss automatisiert Tests für die implementierten Features generieren können. Dies umfasst sowohl Unit-Tests für einzelne Funktionen und Komponenten als auch Integrationstests für Interaktionen zwischen verschiedenen Modulen. Die generierten Tests müssen den Konventionen des jeweiligen Testing-Frameworks entsprechen und eine messbare Code-Coverage erreichen.

**Review und Integration:** Das System muss Merge Requests erstellen und konfigurieren können, einschließlich aussagekräftiger Titel und Beschreibungen, die die implementierten Änderungen dokumentieren und auf das ursprüngliche Issue verweisen. Es muss den Status von CI/CD-Pipelines überwachen und bei erfolgreichen Durchläufen den Merge Request automatisiert schließen können. Bei Pipeline-Fehlern muss das System die Fehlerursache analysieren und geeignete Korrekturmaßnahmen einleiten, beispielsweise durch Anpassung der Tests oder Überarbeitung des Codes. Diese Anforderung vervollständigt den E2E-Workflow und stellt sicher, dass die Implementierungen nicht nur generiert, sondern auch in den Hauptbranch integriert werden.

**Datenerfassung und Metrik-Extraktion:** Das System muss während des gesamten Workflows systematisch Metriken erfassen und persistieren. Dies umfasst zeitliche Metriken wie die Durchlaufzeit einzelner Phasen und des Gesamtworkflows, Erfolgsmetriken wie den Merge-Request-Status und Pipeline-Ergebnisse sowie Prozessmetriken wie die Anzahl der Iterationen, aufgetretene Fehler und deren Kategorien. Die gesammelten Daten müssen in strukturierter Form exportiert werden können, um anschließende statistische Auswertungen zu ermöglichen. Diese Anforderung ist fundamental für die Beantwortung beider Forschungsfragen, da sie die empirische Basis für die Evaluation schafft.

**Fehlerbehandlung und Resilienz:** Das System muss mit transienten Fehlern umgehen können, wie sie bei API-Aufrufen oder LLM-Generierungen auftreten können. Es muss über Retry-Mechanismen verfügen, die bei temporären Ausfällen automatisch weitere Versuche unternehmen. Gleichzeitig muss das System permanente Fehler erkennen und dokumentieren, sodass fehlgeschlagene Simulationen nicht zu ungültigen Datenpunkten führen.

**Konfigurierbarkeit und Parametrisierung:** Das System muss über externe Konfigurationsdateien steuerbar sein, sodass Parameter wie der verwendete LLM-Provider, das Modell, die Temperatur, Random Seeds und weitere Einstellungen ohne Code-Änderungen angepasst werden können. Dies ermöglicht es, verschiedene Konfigurationen systematisch zu vergleichen und die Evaluation unter verschiedenen Bedingungen durchzuführen. Die Konfiguration muss für jeden Durchlauf automatisch dokumentiert werden, sodass die Ergebnisse eindeutig den verwendeten Parametern zugeordnet werden können.

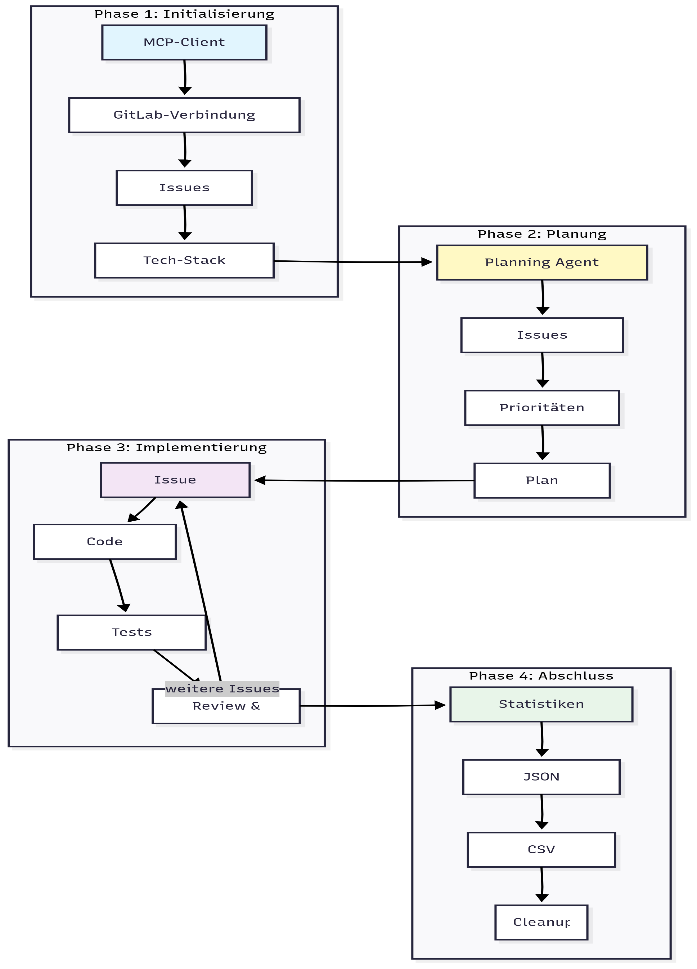
### Out of Scope

**Skalierbarkeit und Robustheit:** Das System ist für die Bearbeitung kleiner bis mittlerer Beispielprojekte mit einer überschaubaren Anzahl von Issues konzipiert. Optimierungen für den industriellen Einsatz mit hunderten parallelen Issues, horizontaler Skalierung oder komplexen Caching-Strategien sind nicht vorgesehen. Die Fehlertoleranz beschränkt sich auf grundlegende Retry-Mechanismen für transiente Fehler. Fortgeschrittene Self-Healing-Mechanismen, Circuit-Breaker-Patterns oder Disaster-Recovery-Strategien würden den Rahmen des Proof-of-Concept sprengen. Das System setzt eine weitgehend stabile Infrastruktur voraus und dokumentiert Fehler, anstatt umfassende automatische Wiederherstellungsversuche zu unternehmen.

**Security und Compliance:** Das System operiert in einer kontrollierten Evaluationsumgebung mit vertrauenswürdigen Eingaben. Eine umfassende Security-Härtung gegen Prompt-Injection-Angriffe, eine rollenbasierte Rechteverwaltung oder die Implementierung von Datenschutzmaßnahmen gemäß DSGVO sind nicht Teil des Scopes. Die GitLab-Integration erfolgt über einen Service-Account mit vollständigen Rechten auf die Template-Projekte. Die Kommunikation mit LLM-Providern nutzt Standard-HTTPS ohne zusätzliche Verschlüsselung oder spezielle Vertragswerke zur Datenverarbeitung. Für die wissenschaftliche Evaluation mit synthetischen Daten sind diese Aspekte nicht erforderlich, wären jedoch für den produktiven Einsatz unabdingbar.

**Monitoring und Ressourcen-Management:** Während das System grundlegende Metriken für die wissenschaftliche Evaluation erfasst, verfügt es nicht über ein umfassendes Monitoring-System mit Integration in Plattformen wie Prometheus oder Grafana, Alerting-Mechanismen oder detailliertes Performance-Profiling. Die Logs dienen der nachträglichen Analyse für Forschungszwecke, nicht der Echtzeitüberwachung. Kostenoptimierung beschränkt sich auf die Dokumentation anfallender LLM-API-Kosten. Strategien wie dynamische Modellauswahl basierend auf Kosteneffizienz, intelligentes Caching redundanter Anfragen oder Budget-Limits mit automatischer Pausierung sind nicht implementiert.

## Lösungskonzept

Das in dieser Arbeit entwickelte Lösungskonzept adressiert die in Kapitel 1.2 identifizierten Forschungslücken durch einen integrierten Ansatz, der drei zentrale Säulen vereint: die vollständige End-to-End-Orchestrierung von GitLab-Workflows, die Standardisierung der Tool-Integration über das Model Context Protocol (MCP) sowie die Kontrolle der Reproduzierbarkeit durch ein Supervisor-basiertes Multi-Agenten-System.

Das Lösungskonzept basiert auf einem mehrstufigen Workflow, der durch einen zentralen Supervisor koordiniert wird:

Abbildung Workflow Phasen

Diese Phasen entsprechen dem in Kapitel 3.3 beschriebenen Versuchsaufbau und ermöglichen die systematische Erfassung der in Kapitel 3.4 definierten Metriken.

## Architekturkonzept

Die Architektur des Systems folgt dem Prinzip der Schichtenbildung (layered architecture), um Komplexität zu reduzieren, Wartbarkeit zu erhöhen und klare Verantwortlichkeiten zu schaffen. Diese Strukturierung ermöglicht zudem eine modulare Testbarkeit der einzelnen Komponenten, was für die wissenschaftliche Evaluation von zentraler Bedeutung ist. Das System gliedert sich in fünf logische Schichten, die aufeinander aufbauen und über definierte Schnittstellen kommunizieren:

**Design-Entscheidung: Schichtenarchitektur**

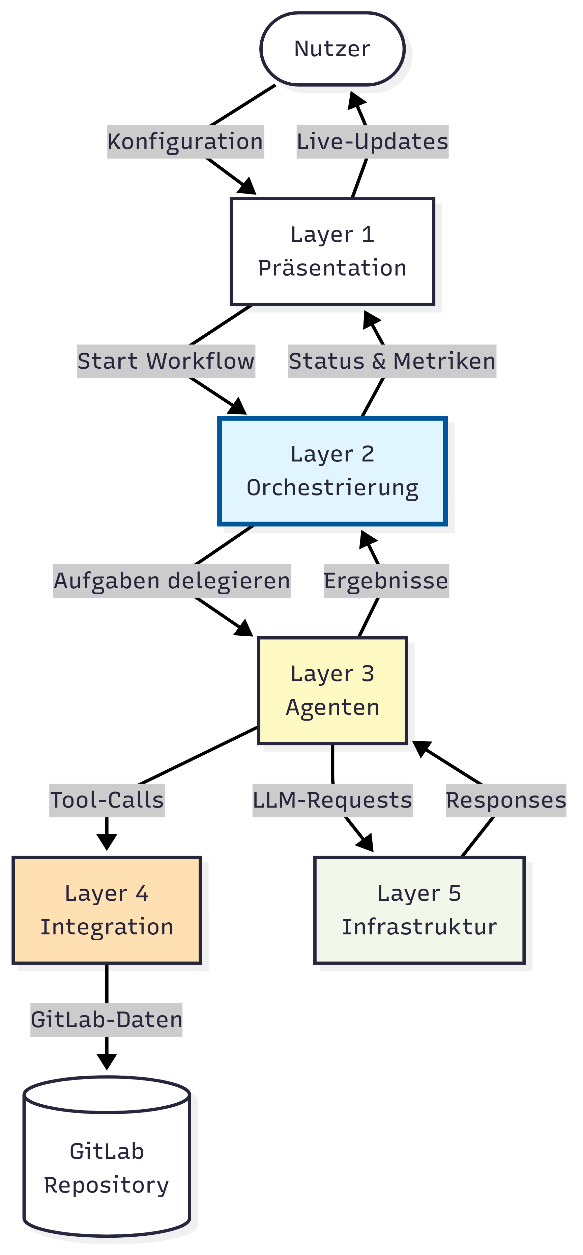
Die Wahl einer Schichtenarchitektur erfolgt aus mehreren Gründen:

Abbildung Schichtenarchitektur

1. Separation of Concerns: Jede Schicht hat eine klar definierte Verantwortlichkeit. Die Präsentationsschicht kümmert sich ausschließlich um Nutzerinteraktion, die Orchestrierungsschicht um Workflow-Koordination, die Agentenschicht um Task-Ausführung, die Integrationsschicht um externe Anbindungen und die Infrastrukturschicht um grundlegende Dienste.
2. Modularität und Austauschbarkeit: Komponenten innerhalb einer Schicht können ausgetauscht werden, ohne andere Schichten zu beeinflussen. Beispielsweise kann die Web GUI durch eine andere Oberfläche ersetzt werden, ohne die Orchestrierungslogik anzupassen.

Diese Architektur-Entscheidung findet Unterstützung in etablierten Software-Engineering-Prinzipien und wird in modernen AI-Systemen zunehmend eingesetzt, um Komplexität zu beherrschen und Nachvollziehbarkeit zu gewährleisten.

## Multi-Agenten-Orchestrierung

Die Orchestrierung mehrerer spezialisierter Agenten bildet das Herzstück des Systems. Im Gegensatz zu monolithischen Ansätzen, bei denen ein einzelner Agent alle Aufgaben bewältigen muss, ermöglicht die Aufteilung in spezialisierte Rollen eine gezielte Prompt-Optimierung, reduzierte Komplexität pro Agent und klarere Fehlerdiagnose.

### Supervisor-Pattern als Koordinationsmodell

Die Literatur unterscheidet zwischen zentralisierter Orchestrierung (ein Supervisor koordiniert alle) und dezentraler Koordination (Agenten handeln autonom und stimmen sich über Kommunikationsprotokolle ab). [37] Für dieses System wurde bewusst die zentrale Orchestrierung gewählt.

Das Supervisor-Pattern bietet entscheidende Vorteile für die wissenschaftliche Evaluation:

**Begründung:**

1. Reproduzierbarkeit (FF.2): Eine zentrale Kontrolle ermöglicht deterministische Abläufe, da die Reihenfolge und Art der Agent-Ausführungen explizit gesteuert werden. Dies ist essentiell für die Messung der Varianz zwischen identischen Runs.
2. Nachvollziehbarkeit: Alle Entscheidungen werden zentral getroffen und protokolliert. Dies vereinfacht die Post-hoc-Analyse und das Debugging.
3. Einfachheit: Dezentrale Koordination erfordert komplexe Kommunikationsprotokolle. Für einen Proof-of-Concept mit vier Agenten ist dies unverhältnismäßig.
4. Fehlerbehandlung: Ein zentraler Supervisor kann Fehler einzelner Agenten abfangen, loggen und Recovery-Strategien einleiten, ohne dass andere Agenten davon betroffen sind.

### ReAct-Agent

Das ReAct-Pattern (Reasoning and Acting) ist ein bewährter Ansatz für LLM-Agenten, der explizites Reasoning mit Tool-Verwendung verbindet. [33]

Alternative Patterns wie Plan-and-Execute, Function-Calling-Only oder Prompt-Chaining wurden evaluiert. Die Entscheidung für ReAct basiert auf folgenden Überlegungen:

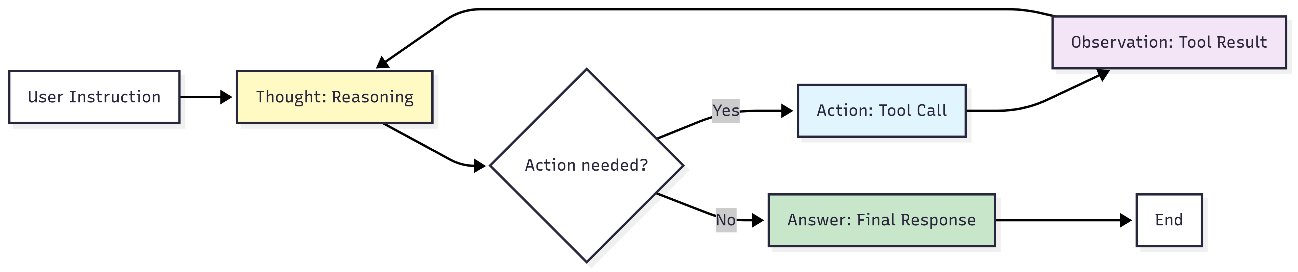
1. Explizites Reasoning → Nachvollziehbarkeit: ReAct-Agenten verbalisieren ihren Denkprozess („Thought"). Dies ermöglicht es, die Entscheidungslogik zu verstehen und zu validieren – ein zentrales Kriterium für wissenschaftliche Evaluation.
2. Tool-Calling-Integration: ReAct integriert nahtlos Function-Calling für GitLab-Operationen. Der Agent kann zwischen Reasoning und Tool-Use wechseln, ohne den Kontext zu verlieren.
3. Bewährtes Pattern mit Framework-Support: LangGraph bietet vorgefertigte ReAct-Implementierungen (create\_react\_agent), was die Entwicklung beschleunigt und auf erprobte Komponenten zurückgreift.
4. Flexibilität: Im Gegensatz zu starren Plan-and-Execute-Patterns kann ReAct dynamisch auf unerwartete Tool-Outputs reagieren und den Plan anpassen.

Abbildung ReAct-Pattern

### Spezialisierte Agentenrollen

Anstatt einen generischen Agent für alle Aufgaben zu verwenden, setzt das System auf Spezialisierung. Die Literatur zeigt, dass spezialisierte Prompts zu konsistenteren Ergebnissen führen als generische Instruktionen. Zudem reduziert Spezialisierung die Prompt-Komplexität pro Agent und ermöglicht klare Erfolgskriterien. Die Rollenaufteilung orientiert sich an klassischen Software-Engineering-Phasen und dem GitLab-Workflow. [8]

**Konzept der Spezialisierung:**

Jeder Agent erhält eine klar definierte Rolle, einen spezifischen System-Prompt, Zugriff auf relevante Tools und eine definierte Output-Struktur. Dies reduziert die Komplexität pro Agent und ermöglicht gezielte Optimierung.

**Agentenrollen im Detail:**

1. **Planning Agent**
   * **Verantwortung:** Analyse der GitLab-Issues, Priorisierung nach Kritikalität und Abhängigkeiten, Erstellung eines strukturierten Implementierungsplans
   * **Input:** Liste offener Issues, Tech-Stack-Information, Projekt-Kontext
   * **Output:** JSON-strukturierter Plan mit priorisierten Issues
   * **Tools:** list\_issues, get\_project, get\_file (für Kontext-Analyse)
2. **Coding Agent**
   * **Verantwortung:** Generierung von Code-Lösungen für Issues, Branch-Erstellung, File-Modifikationen, Commit-Erstellung
   * **Input:** Issue-Beschreibung, Tech-Stack, bestehende Code-Struktur
   * **Output:** Code-Änderungen, Branch-Name, Commit-Message
   * **Tools:** create\_branch, get\_file, update\_file, create\_commit
3. **Testing Agent**
   * **Verantwortung:** Generierung von Unit- und Integrationstests, Validierung der Code-Coverage, Trigger der CI/CD-Pipeline
   * **Input:** Implementierter Code, Test-Framework (pytest, jest, etc.)
   * **Output:** Test-Files, Coverage-Report, Pipeline-ID
   * **Tools:** update\_file (Tests schreiben), create\_commit, create\_merge\_request (triggert Pipeline)
4. **Review Agent**
   * **Verantwortung:** Validierung des Merge Requests, Prüfung des Pipeline-Status, Merge-Entscheidung
   * **Input:** Merge Request, Pipeline-Status, Code-Diff
   * **Output:** Review-Kommentar, Merge/Reject-Entscheidung
   * **Tools:** get\_merge\_request, Pipeline-Status-Abfrage (über MR), approve\_merge\_request

### Kommunikations- und Koordinationskonzept

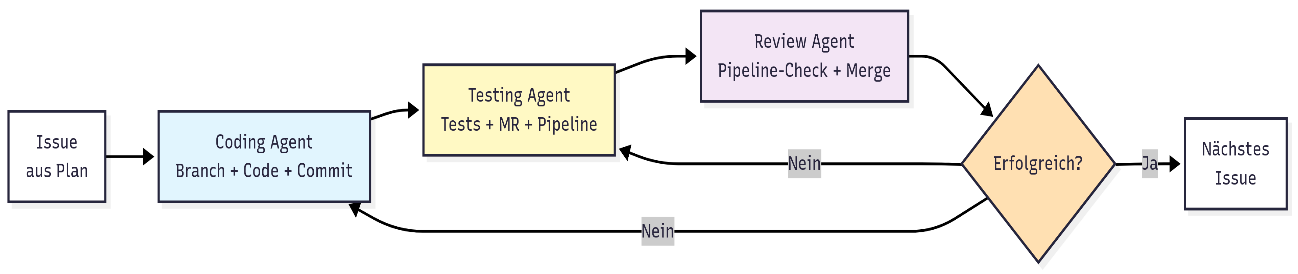
Ein zentraler Aspekt der Agenten-Orchestrierung ist die Frage, ob Agenten sequentiell [11] oder parallel [42] ausgeführt werden sollen. Das System nutzt eine streng sequentielle Ausführung der Agenten pro Issue:

Abbildung Sequentieller Issue Prozess

1. Determinismus für Evaluation: Sequentielle Ausführung eliminiert Race-Conditions und nicht-deterministische Interaktionsreihenfolgen. Dies ist entscheidend für die Reproduzierbarkeit (FF.2).
2. State-Konsistenz: Jeder Agent kann sich darauf verlassen, dass der vorherige Agent seine Aufgabe vollständig abgeschlossen hat. Der Testing Agent weiß, dass der Code bereits committed wurde; der Review Agent weiß, dass Tests existieren.
3. Vereinfachte Fehlerdiagnose: Bei paralleler Ausführung können Fehler aus Interaktionen zwischen Agenten entstehen, die schwer zu diagnostizieren sind. Sequentiell ist die Fehlerquelle eindeutig.
4. Reduzierte Komplexität: Synchronisation und Koordination bei paralleler Ausführung erfordern zusätzliche Mechanismen (Locks, Message Queues, etc.), die für einen Proof-of-Concept unverhältnismäßig sind.

Sequentielle Ausführung ist langsamer, aber deutlich kontrollierbarer. Für eine wissenschaftliche Evaluation, bei der Nachvollziehbarkeit und Reproduzierbarkeit im Vordergrund stehen, ist dies der richtige Ansatz.

## Prompt-Design

Das Prompt-Design ist kritisch für konsistente Agent-Ergebnisse und klare abgrenzungen zu abhängigkeiten und aufgaben sowie Grenzen. [16]

**System-Prompt-Struktur:**

1. Rollendeklaration ("You are a {Agent-Type} specialized in...")
2. Verantwortlichkeiten (klare Auflistung)
3. Tech-Stack-Context (Projekt-spezifisch, dynamisch injiziert)
4. Verfügbare Tools (Liste mit Beschreibungen)
5. Output-Format-Vorgabe (z.B. JSON für Planning Agent)
6. Completion-Marker-Instruktion

**Begründung**

* Konsistenz: Identische Prompts über Runs hinweg [34]
* Reproduzierbarkeit: Templates sind versioniert und dokumentiert
* Anpassbarkeit: Tech-Stack wird dynamisch injiziert

Woher weiß der Supervisor, dass ein Agent seine Aufgabe abgeschlossen hat?

|  |  |
| --- | --- |
| **Agent** | **Primäre Marker** |
| Planning | "PLAN COMPLETE" |
| Coding | "IMPLEMENTATION COMPLETE" |
| Testing | "TESTS GENERATED" |
| Review | "MR APPROVED" |

Tabelle 2 Multi-Pattern Success-Marker-System

## Workflow-Design

Der End-to-End-Workflow gliedert sich in vier Hauptphasen, die vom Supervisor koordiniert werden und den in Kapitel 3.3 beschriebenen Versuchsaufbau operationalisieren.

### Initialisierungsphase

MCP-Client starten (GitLab-Server, FileSystem-Server)

GitLab-Verbindung herstellen und validieren

Issues abrufen

### Planungphase

Planning Agent analysiert alle offenen Issues

Priorisierung nach Labels (critical > high > medium > low)

Abhängigkeitsauflösung (wenn Issue A in Issue B erwähnt wird)

Output: Priorisierte Liste von Issues zur Bearbeitung

### Implementierungsphase

Sequentielle Bearbeitung jedes Issues

Pro Issue: Coding → Testing → Review (3-Phasen-Workflow)

Metriken-Erfassung während der Bearbeitung

### Abschluss

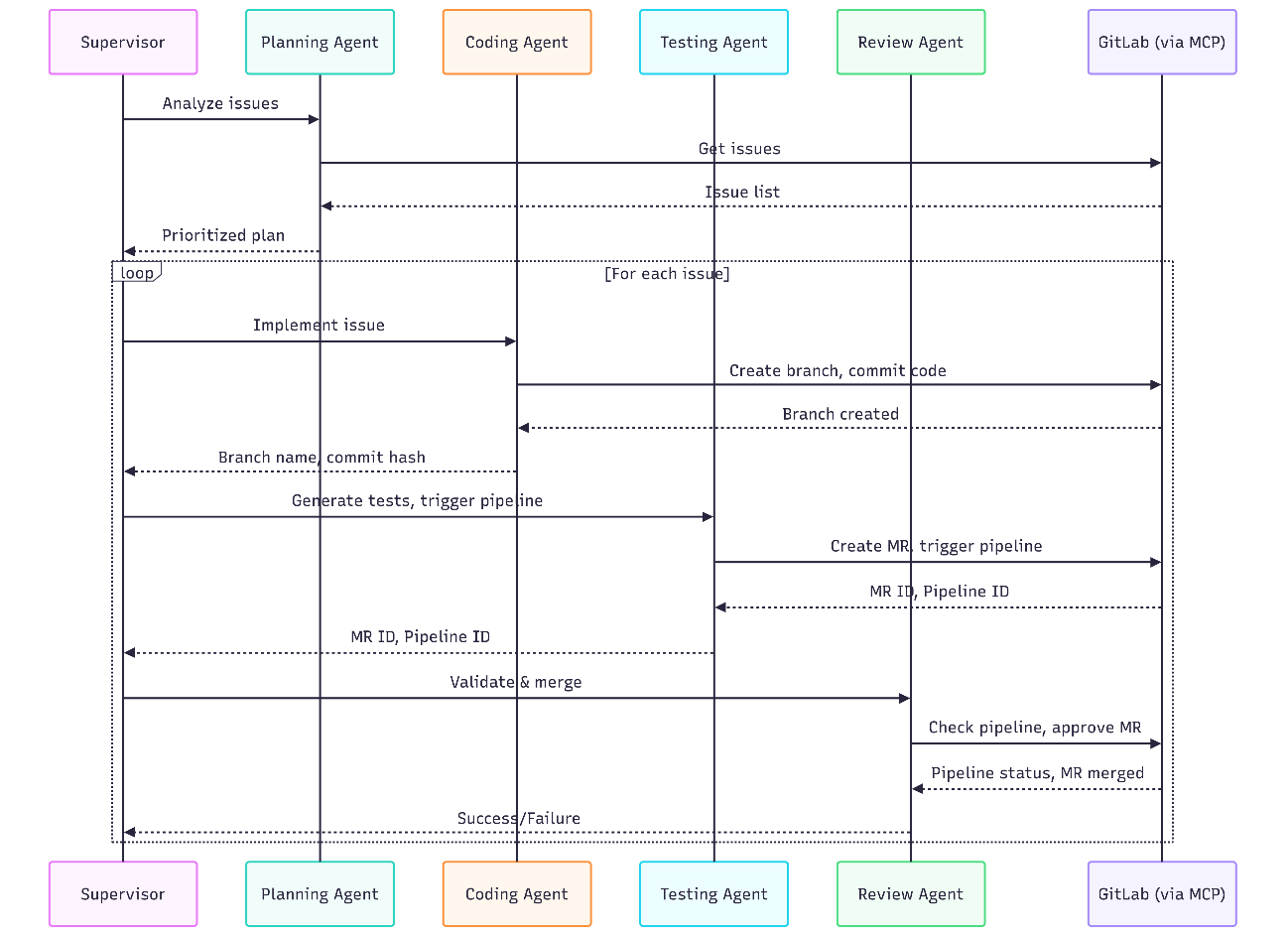
Aggregation aller gesammelten Metriken

Export in zwei Formaten: JSON (detailliert) und CSV (aggregiert für Statistik)

Cleanup und Shutdown

## Datenfluss

Agenten kommunizieren indirekt über den Supervisor, nicht direkt untereinander. Der Datenfluss pro Issue:



## Messkonzept

Die Operationalisierung der Metriken (Kapitel 3.4) erfolgt durch systematische Datenerfassung während des Workflows.

### Erfolgsmetriken

**Operationalisierung**

| **Metrik** | **Datenquelle** | **Erhebung** | **Codierung** |
| --- | --- | --- | --- |
| **MR-Erfolg** | GitLab API | MR.state == 'merged' | Binär (1/0) |
| **Pipeline-Erfolg** | GitLab API | Pipeline.status == 'success' | Binär (1/0) |
| **Bugfix-Korrektheit** | Manuelle Validierung | Tests bestanden? | Binär (1/0) |

**Granularität**

Issue-Level: Erfolg/Fehlschlag pro Issue

Run-Level: Erfolgsrate = successful\_issues / total\_issues

### Prozessmetriken

| **Metrik** | **Messung** | **Granularität** |
| --- | --- | --- |
| **Durchlaufzeit** | end\_time - start\_time (Timestamps) | Pro Phase, Pro Issue, Pro Run |
| **Fehlerrate** | failed\_attempts / total\_attempts | Tool-Calls, LLM-Requests, Pipelines |
| **Review-Zyklen** | Zähler im Agent-Executor | Pro Issue |

Durchlaufzeit: Effizienz-Indikator

Fehlerrate: Robustheit-Indikator

Review-Zyklen: Komplexitäts-Indikator (mehr Zyklen = schwierigere Issues)

### Metriken-Speicherung-Konzept

Zwei Formate für verschiedene Zwecke:

1. JSON (detailliert):

* Vollständige Run-Metadaten
* Alle Issues mit Agent-Outputs
* Zweck: Detailanalyse, Debugging, Reproduzierbarkeit

1. CSV (aggregiert):

* Tabellarische Zusammenfassung
* Eine Zeile pro Issue/Pipeline/Agent
* Zweck: Statistische Auswertung (Kapitel 6)

# Implementierung

Nachdem in Kapitel 4 das konzeptionelle Design des Artefakts dargelegt wurde, beschreibt dieses Kapitel die konkrete technische Umsetzung. Dabei liegt der Fokus nicht auf einer vollständigen Code-Dokumentation, sondern auf der Darstellung technischer Herausforderungen, die während der Implementierung auftraten, sowie der entwickelten Lösungsansätze. Die Implementierung folgt dem in Kapitel 3 beschriebenen Design-Science-Research-Ansatz und validiert die Machbarkeit des in Kapitel 4 vorgestellten Designs.

## Entwicklungszyklen

Die Entwicklung erfolgte in zwei Monaten als iterativer Prozess mit kurzen Bausteinen (Planung → Implementierung → Messung → Nachjustierung). Diese Vorgehensweise war notwendig, weil das Interaktionsverhalten von LLM-Agenten in realen Tool-Ketten (Git, Pipelines, MRs) nur begrenzt a priori prognostizierbar ist und wiederholte Testruns benötigt, um Fehlermuster aufzudecken (z. B. falsche Interpretation von Pipeline-Zuständen, unzulässige Merge-Entscheidungen). Jeder Zyklus schloss mit einer Review-Session ab, in der Prompt-Instruktionen, Tool-Schemata und Kontrolllogik (Guardrails) verfeinert wurden.

## Technologie-Stack

Die Auswahl der verwendeten Technologien erfolgte auf Basis der in Kapitel 4 definierten Anforderungen sowie bewährter Praktiken aus der Softwareentwicklung und dem Machine-Learning-Engineering. Tabelle 3 gibt einen Überblick über die zentralen Komponenten des Technology-Stacks.

| **Kategorie** | **Technologie** | **Version** | **Begründung** |
| --- | --- | --- | --- |
| Programmiersprache | Python | 3.13 | Native Async/Await-Unterstützung, Type Hints für Wartbarkeit, reiches KI-Ökosystem mit etablierten Bibliotheken |
| LLM-Framework | LangChain / LangGraph | 0.2.x | State-of-the-Art für ReAct-Agenten, integrierte Tool-Unterstützung, Community-Support, prebuilt Agent-Templates |
| GitLab-Integration | Model Context Protocol (MCP) | 1.0 | Standardisierte, auditierbare Schnittstelle (Kapitel 4.6), Portabilität über LLM-Clients hinweg |
| Web-Framework | FastAPI | 0.115.x | Native Async-Unterstützung, WebSocket-Integration, automatische API-Dokumentation via OpenAPI |
| Testing-Framework | pytest + pytest-asyncio | 8.x, 1.2.x | De-facto-Standard für Python-Testing, robuste Async-Test-Unterstützung |
| Metriken & Logging | Python Dataclasses + JSON | Standard-Lib | Einfachheit, direkte Serialisierung, keine externen Abhängigkeiten |
| Versionskontrolle | Git + GitHub | - | Basis für E2E-Workflow (Kapitel 2.1), integrierte CI/CD |

Die Wahl fiel auf LangChain/LangGraph als Framework für die Agent-Implementierung. Diese Entscheidung basierte auf mehreren Faktoren:

* Breite Verwendung in ähnlichen Projekten: LangGraph hat sich als De-facto-Standard für Multi-Agent-Systeme mit LLMs etabliert, was die wissenschaftliche Vergleichbarkeit erhöht
* Vorgefertigte ReAct-Implementierung: Das Framework bietet create\_react\_agent(), das das in Kapitel 4.5.1 beschriebene ReAct-Pattern bereits implementiert
* Provider-Abstraktion: Native Unterstützung für verschiedene LLM-Provider (OpenAI, DeepSeek, Ollama) über einheitliche Schnittstelle
* Community und Dokumentation: Aktive Community und umfassende Dokumentation reduzieren Entwicklungsrisiko

Die Verwendung von LangGraph brachte jedoch auch Herausforderungen mit sich:

Problem 1: Tool-Calling-Formate

Verschiedene LLM-Provider verwenden unterschiedliche Formate für Function-Calling. Während OpenAI-konforme LLMs (DeepSeek, Ollama mit OpenAI-kompatiblen Modellen) problemlos funktionierten, traten bei Gemini (Google) Inkompatibilitäten auf:

**Erwartung**: Einheitliches Tool-Calling über LangGraph-Abstraktion

**Realität**: Gemini verwendet anderes JSON-Schema für Tool-Calls

**Lösung**: Fokussierung auf OpenAI-konforme Provider für wissenschaftliche Evaluation

Die mangelnde Standardisierung von Tool-Calling-Formaten zwischen LLM-Providern ist eine Herausforderung für Provider-unabhängige Systeme. Das Model Context Protocol (MCP) adressiert dies teilweise, aber Provider-spezifische Anpassungen bleiben notwendig.

LangGraph's Graph-basierte Architektur (State Graphs, Message Graphs) erforderte signifikante Einarbeitung. Die Dokumentation ist umfassend, aber die Abstraktion ist komplex für einfache sequentielle Workflows.

Trotz der Herausforderungen erwies sich LangGraph als richtige Wahl:

* Zeit-Ersparnis: Vorgefertigte Patterns reduzierten Entwicklungszeit erheblich
* Robustheit: Framework-basierter Error-Handling übertraf Custom-Implementierung
* Komplexität: Für einfache sequentielle Workflows möglicherweise over-engineered
* Provider-Lock-in: De-facto auf OpenAI-konforme LLMs beschränkt

## Implementierungsschwerpunkt: Prompt-Engineering

Die Prompts sind der größte Hebel für **Zuverlässigkeit** und **Reproduzierbarkeit** für die Agenten. Während das Design in Kapitel 4.6.1 Template-basierte Prompts vorsah, erwies sich die Entwicklung funktionierender Prompts als kontinuierlicher, iterativer Prozess.

### Herausforderung: LLM-Arbeit mit GitLab gesteuert durch Prompts

Die Steuerung von LLM-Agenten zur Interaktion mit GitLab über Prompts erwies sich als **durchgehendes Problem**, das kontinuierliches Finetuning erforderte.

**Kernprobleme:**

1. **Pipeline-Überwachung wurde übergangen**
   * Review-Agent sollte Pipeline-Status validieren (Kapitel 4.4.2)
   * **Problem:** Agent übersprang Validierung oder interpretierte Status falsch
   * **Beispiel:** Agent meldet "PIPELINE SUCCESS" obwohl GitLab-API status: 'running' zurückgibt
2. **Merge trotz fehlgeschlagener Pipeline**
   * **Kritischster Fehler:** Review-Agent merged MR obwohl Pipeline fehlgeschlagen war
   * **Agent-Begründung:** "Previous pipeline was successful, current failure is transient"

### Iterativer Prompt-Engineering-Prozess

Die Lösung dieser Probleme erfolgte durch systematisches, datengetriebenes Prompt-Engineering:

1. Problem-Identifikation: Analyse fehlgeschlagener Runs, Extraktion von Failure-Patterns
2. Hypothesenbildung: Vermutung über Ursache (z.B. "Agent übersieht Pipeline-Status in JSON")
3. Prompt-Anpassung: Gezielte Modifikation (z.B. explizite Instruktion zur Status-Validierung)
4. Validierung: Test-Runs mit modifiziertem Prompt
5. Iteration: Bei Nicht-Erfolg zurück zu Schritt 2

**Beispiel: Review-Agent Pipeline-Validierung**

**Iteration 1:**

“You are a Review Agent. Validate the pipeline and merge if successful.

Tools: check\_pipeline\_status, approve\_merge\_request”

**Problem:** Agent merged ohne Pipeline-Prüfung

**Iteration 2:**

“You are a Review Agent.

CRITICAL INSTRUCTIONS:

1. Use check\_pipeline\_status tool to get pipeline status

2. Only approve if status == "success"

3. If status == "failed", DO NOT merge

Available tools: check\_pipeline\_status, approve\_merge\_request”

**Problem**: Agent merged bei status="running" mit Begründung "Pipeline wird erfolgreich sein"

**Iteration 3 (Final):**

You are a Review Agent for GitLab Merge Requests.

STRICT VALIDATION PROTOCOL:

1. Call check\_pipeline\_status(pipeline\_id)

2. Extract "status" field from JSON response

3. Valid status values: "success", "failed", "running", "canceled"

DECISION RULES (NO EXCEPTIONS):

- IF status == "success" → approve\_merge\_request()

- IF status == "failed" → Report failure, DO NOT merge

- IF status == "running" → Wait and recheck

- IF status == "canceled" → Report failure, DO NOT merge

FORBIDDEN ACTIONS:

- NEVER merge without calling check\_pipeline\_status first

- NEVER interpret partial success as success

- NEVER merge if status != "success"

Complete your task by stating: "VALIDATION COMPLETE: [APPROVED|REJECTED]"

Das konnte auf alle Prompts dementsprechend angewendet werden:

1. Explizitheit > Implizitheit: Je expliziter die Instruktionen, desto konsistenter die Outputs [33]
2. Negativ-Instruktionen wichtig: "DO NOT"-Anweisungen reduzieren unerwünschtes Verhalten
3. Strukturierte Decision-Trees: IF-THEN-Regeln reduzieren Interpretationsspielraum
4. Completion-Marker essentiell: Explizite End-Marker erhöhen Erkennungsrate (Kapitel 4.6.2)

### Prompt-Template-Struktur

Nach iterativer Verfeinerung kristallisierte sich folgende Best-Practice-Struktur heraus:

1. Rollendeklaration (1-2 Sätze)

2. Kontext (Projekt-Info, Tech-Stack)

3. CRITICAL INSTRUCTIONS (Hauptaufgabe)

4. STRICT VALIDATION/DECISION RULES (IF-THEN-Logik)

5. FORBIDDEN ACTIONS (explizite Verbote)

6. Available Tools (Liste mit Beschreibungen)

7. Output Format (JSON-Schema oder Text-Pattern)

8. Completion Marker (explizites End-Signal)

## Implementierungsschwerpunkt: Orchestrierung

Der zweite zentrale Fokus lag auf der Orchestrierung des Multi-Agenten-Workflows durch den Supervisor.

Der Supervisor implementiert die Phasen-Transitions:

1. **INITIALIZING-Phase:**
   * MCP-Client-Setup (GitLab + FileSystem Server)
   * Tech-Stack-Detection (automatisch oder konfiguriert)
   * Issue-Fetching
   * Transition zu PLANNING bei Erfolg
2. **PLANNING-Phase:**
   * Aufruf Planning-Agent
   * Plan-Parsing und Validierung
   * Transition zu IMPLEMENTING
3. **IMPLEMENTING-Phase:**
   * **Sequentielle Iteration** über alle geplanten Issues
   * Pro Issue: Coding → Testing → Review (3-Phasen-Workflow)
   * Transition zu COMPLETING nach letztem Issue
4. **COMPLETING-Phase:**
   * Metriken-Aggregation
   * JSON/CSV-Export
   * Transition zu COMPLETED

### Sequentielle Koordination

Design-Vorgabe (Kapitel 4.3.3): Sequentielle Agent-Ausführung pro Issue

Implementierung:

Der AgentExecutor (src/orchestrator/core/agent\_executor.py)

## A diagram of a company AI-generated content may be incorrect.Ergebnis der Implementation

A diagram of a project

AI-generated content may be incorrect.

# Evaluation & Experimente

Die in Kapitel 3 formulierten Forschungsfragen (FF.1 und FF.2) sowie die daraus abgeleiteten Hypothesen bilden den Ausgangspunkt der Evaluation. Ziel ist es, den in Kapitel 4 entworfenen und in Kapitel 5 implementierten Prototyp unter realitätsnahen, aber kontrollierten Bedingungen empirisch zu testen. Im Mittelpunkt stehen zwei Dimensionen:

1. der Einfluss der **syntaktischen Anforderungsqualität** auf die Performanz von LLM-Agenten in GitLab-Workflows (FF.1) und
2. die **Robustheit und Reproduzierbarkeit** der Ergebnisse bei wiederholten Runs unter identischen Parametern (FF.2).

Die Evaluation dient damit dem Nachweis, dass die entwickelte Simulationsumgebung in der Lage ist, GitLab-basierte Softwareentwicklungsprozesse vollständig Ende-zu-Ende zu orchestrieren und dabei reproduzierbare, messbare Ergebnisse zu liefern.

## Planung & Aufbau der Experimente

Für die Evaluation werden zwei GitLab-Projekt-Templates erstellt, die sich ausschließlich in der syntaktischen Qualität der formulierten Anforderungen unterscheiden. Alle weiteren Parameter – Technologie-Stack, CI/CD-Konfiguration, Repository-Struktur und Agenten-Setup – werden konstant gehalten, um einen kontrollierten Vergleich zu ermöglichen.

* Projekt A repräsentiert eine Umgebung mit hoher Anforderungsqualität, formuliert nach den Kriterien der Norm ISO/IEC/IEEE 29148:2018 (Eindeutigkeit, Konsistenz, Testbarkeit, Umsetzbarkeit).
* Projekt B spiegelt hingegen niedrige Anforderungsqualität wider, bei der die gleichen fachlichen Inhalte bewusst mehrdeutig, redundant oder unstrukturiert beschrieben sind.

Diese Differenzierung erlaubt es, den Einfluss sprachlicher Qualität auf die Performanz der LLM-Agenten empirisch zu messen, ohne dass technische Faktoren das Ergebnis verzerren.

### Projektauswahl

Als Basis für beide Templates wird ein Java-basiertes Maven-Projekt gewählt.

Die Entscheidung für Java beruht auf mehreren Gründen:

* Etablierte Sprache im Software Engineering: Java ist in der Industrie weit verbreitet, insbesondere in Kombination mit GitLab CI/CD. Dadurch ist die Simulation praxisnah und übertragbar auf reale Szenarien.
* Klar definierter Build-Prozess: Maven bietet eine standardisierte Projektstruktur, automatisierte Dependency-Verwaltung und klar getrennte Build-Phasen (compile, test, package).
* Kompatibilität mit GitLab-Runnern: Die Kombination aus Maven und GitLab ermöglicht eine einfache Integration, ohne komplexe Setups oder manuelle Skripte.
* Reduzierte Störvariablen: Im Gegensatz zu dynamischen Sprachen wie Python führt Java bereits bei Kompilierungsfehlern zu klaren Metriken, die die Performanz des Coding-Agents messbar machen.

Das Ziel der Evaluation ist nicht die Entwicklung neuer Build-Prozesse, sondern die Untersuchung, wie der Prototyp in realen, reproduzierbaren GitLab-Workflows agiert.

### Pipeline aufbau

Beide Projekt-Templates nutzen eine identische, bewusst einfach gehaltene GitLab CI/CD-Pipeline, um die Agentenleistung isoliert zu evaluieren.

„

PIPELIE-Config?

„

**Erläuterung der Pipeline-Entscheidung:**

* Die Pipeline enthält nur zwei zentrale Phasen (Compile und Test), um die Agenteninteraktionen fokussiert zu evaluieren.
* Der **Coding-Agent** wird primär anhand der Kompilierbarkeit des Codes bewertet.
* Der **Testing-Agent** wird anhand der generierten Tests und deren Coverage beurteilt.
* Das **Testziel** liegt bei einer **Coverage von ≥ 70 %**, um eine solide Codequalität sicherzustellen.
* Als Runner wird der dedizierte „SESL“-Runner eingesetzt, der eine isolierte Umgebung mit stabilen Laufzeiten gewährleistet.

Damit lassen sich Metriken wie *Pipeline-Erfolg*, *Fehlerrate* und *Durchlaufzeit* standardisiert erfassen.

### Projekt-Template A

Projekt A enthält Issues mit präzise formulierten User Stories nach ISO/IEC/IEEE 29148 sowie der INVEST-Heuristik. Die Anforderungen sind eindeutig, testbar und konsistent aufgebaut.

**Beispiel-Issue (Projekt A):**

**Titel**: Implementierung einer Validierung für Benutzer-Input

**Beschreibung**:

Als Benutzer möchte ich, dass das System meine Eingaben auf Ungültigkeiten prüft, damit Fehler verhindert werden.

**Akzeptanzkriterien**:

* Leere Eingaben führen zu einer Fehlermeldung.
* Ungültige Zeichen werden abgelehnt.
* Erfolgreiche Eingaben werden gespeichert.
* Testbarkeit: Unit-Tests müssen alle Eingabevarianten abdecken.

Die Anforderungen folgen klaren Satzstrukturen („Als … möchte ich … damit …“) und enthalten überprüfbare Akzeptanzkriterien.

### Projekt-Template B

Projekt B enthält semantisch identische, aber syntaktisch schwache und unstrukturierte Anforderungen, die typische Probleme aus der Praxis widerspiegeln – etwa Mehrdeutigkeiten, fehlende Prüfkriterien, redundante Aussagen und uneinheitliche Satzkonstruktionen. Während die zugrunde liegende fachliche Information unverändert bleibt, wird sie in einer Weise formuliert, die Interpretationsspielraum lässt und dadurch die semantische Belastbarkeit der LLM-Agenten testet.

**Beispiel-Issue (Projekt B):**

**Titel**: Prüfen, dass Eingaben funktionieren

**Beschreibung**:

Wenn jemand etwas eingibt, soll das System schauen, ob das irgendwie passt.

Es darf dabei nichts schiefgehen, aber es soll auch nicht zu streng sein.

Es wäre gut, wenn Eingaben, die eigentlich richtig sind, auch funktionieren.

Fehler sollten am besten nicht passieren, aber das System sollte trotzdem weitermachen.

**Anmerkung**: Vielleicht braucht man Tests dafür, aber das muss man sehen, wenn es soweit ist.

Dieses Issue enthält inhaltlich dieselben Informationen wie das Beispiel aus Projekt A die Notwendigkeit einer Eingabevalidierung und der Vermeidung von fehlerhaften Benutzereingaben formuliert jedoch auf eine unscharfe, redundante und nicht überprüfbare Weise.

* Die eigentlichen Akzeptanzkriterien fehlen oder sind nur implizit enthalten („Fehler sollten am besten nicht passieren“).
* Es existieren keine klaren Bedingungen oder erwarteten Ergebnisse, was eine objektive Testbarkeit verhindert.
* Der sprachliche Stil ist uneinheitlich („irgendwie“, „am besten“, „vielleicht“), was die semantische Verarbeitung durch LLMs erschwert.

Damit bietet Projekt B die notwendige Grundlage, um empirisch zu untersuchen, wie sich syntaktische und semantische Unschärfen in Anforderungen auf den Agenten-Workflow auswirken insbesondere auf die Qualität der generierten Code-Implementierungen, die Erfolgsquote von CI/CD-Pipelines und die Varianz der Ergebnisse zwischen identischen Runs.

## Durchführung

Die Durchführung der Experimente dient sowohl der quantitativen Beantwortung von FF.1 als auch der Varianzbetrachtung gemäß FF.2. Für FF.1 werden zwei Projektszenarien (A: hohe Qualität, B: niedrige Qualität) jeweils zehnmal ausgeführt. Die erhobenen Metriken (Merge-Erfolg, Pipeline-Erfolg, Durchlaufzeit, Fehlerrate, Review-Zyklen) werden anschließend zwischen beiden Projekten verglichen. Dieser Vergleich erlaubt Rückschlüsse auf den Einfluss klarer, konsistenter Anforderungen auf die Leistungsfähigkeit der Agenten.

Zur Beantwortung der Forschungsfragen werden die Simulationen nach einem festen Schema durchgeführt:

|  |  |
| --- | --- |
| Parameter | Beschreibung |
| Runs pro Projekt | 10 Runs für Projekt A und 10 für Projekt B |
| LLM-Modell | Deepseek coder, Temperatur = 0,2 |
| Agenten-Setup | Supervisor + vier Rollenagenten (Planning, Coding, Testing, Review) |
| GitLab Umgebung | Gleiche Runner-, Repository- und Pipeline-Konfiguration |
| Datenerfassung | Automatische Protokollierung über GitLab Agenten-Logs |

## Bewertung und Analyse

Zur Beantwortung der ersten Forschungsfrage wird die mittlere Performanz der Agenten in beiden Projekten verglichen. Erwartet wird, dass Projekt A signifikant bessere Erfolgs- und Prozessmetriken erzielt, da klar strukturierte Anforderungen die semantische Interpretation durch LLMs erleichtern.

**Auswertungsschritte:**

* Berechnung der Mittelwerte (𝑥̄) für alle Metriken über 10 Runs.
* Gegenüberstellung der Differenzen (Δ𝑥̄) zwischen A und B.
* Darstellung in Balken- und Boxplots, um Abweichungen und Streuungen sichtbar zu machen.
* Interpretation im Hinblick auf Hypothese H1 („Höhere Anforderungsqualität führt zu besseren Performanzmetriken“).

Für die zweite Forschungsfrage (FF.2) wird untersucht, **wie stark die Ergebnisse bei wiederholten Durchläufen** mit identischen Parametern streuen und ob das System reproduzierbar arbeitet.  
Die Varianz entsteht hauptsächlich durch **stochastische Faktoren** wie LLM-Sampling, API-Latenzen und Modellinterne Nichtdeterminismen.

Ein entscheidender Parameter ist die **Temperatur** des LLM, welche die Zufälligkeit der Textgenerierung steuert:

* **Niedrige Werte (0.0–0.2):** nahezu deterministische, stabile Ausgaben
* **Mittlere Werte (0.4–0.6):** ausgewogene, aber variablere Ergebnisse
* **Hohe Werte (≥ 0.8):** stark divergierende Antworten mit geringer Reproduzierbarkeit

Für diese Evaluation wird **Temperatur = 0.2** gewählt, um zufällige Effekte zu minimieren und den Einfluss der Anforderungsqualität isoliert zu betrachten.

Die Reproduzierbarkeit wird anhand folgender **Kennzahlen** bewertet:

* **Standardabweichung (σ)** der Erfolgsmetriken
* **Varianz (V)** der Durchlaufzeiten
* **Relative Abweichung (Δ%)** zwischen Runs

Ein System gilt als **reproduzierbar**, wenn die Standardabweichung der zentralen Metriken (z. B. Erfolgsquote, Pipeline-Erfolg) ≤ 10 % beträgt.  
Abweichungen darüber deuten auf erhöhte stochastische Einflüsse oder unzureichend kontrollierte Agenteninteraktionen hin.

Insgesamt erlaubt diese Analyse, die **Robustheit und Stabilität** der Simulationsumgebung quantitativ zu bewerten und zu prüfen, ob trotz stochastischer Komponenten eine konsistente Reproduktion der Ergebnisse möglich ist.

### Auswertung: Varianz der Projekte Durchläufe(FF.2)

### Gegenüberstellung: Projekt Ergebnisse (FF.1)

### Zusätzliche Beobachtungen

# Diskussion

Die in Kapitel 6 vorgestellten Evaluationsergebnisse werden in diesem Kapitel interpretiert, kritisch reflektiert und in den Kontext der Forschungsfragen eingeordnet. Dabei werden zunächst die gewonnenen Erkenntnisse und deren Bedeutung für die Praxis des KI-gestützten Software Engineerings diskutiert (Abschnitt 7.1). Anschließend werden methodische und technische Limitationen (Abschnitt 7.2) sowie die im Entwicklungsprozess identifizierten Herausforderungen (Abschnitt 7.3) betrachtet. Abschließend werden weiterführende Forschungsperspektiven und mögliche Erweiterungen der Simulationsumgebung vorgestellt (Abschnitt 7.4).

## Interpretation

Die Evaluation verfolgte zwei zentrale Zielrichtungen:

die Untersuchung des Einflusses der Anforderungsqualität auf die Performanz von LLM-Agenten (FF.1)

die Beurteilung der Reproduzierbarkeit und Varianz der Ergebnisse unter identischen Bedingungen (FF.2).

Die Ergebnisse zeigen deutlich, dass syntaktisch klare und strukturiert formulierte Anforderungen zu einer signifikant besseren Agentenleistung führen. In Projekt A (hohe Anforderungsqualität) erzielten die Agenten durchweg höhere Erfolgsraten in den CI/CD-Pipelines, geringere Fehlerraten und konsistentere Durchlaufzeiten im Vergleich zu Projekt B. Damit wird die Hypothese bestätigt, dass präzise Anforderungen die semantische Verarbeitung durch LLMs erleichtern und so die Gesamtperformanz des Entwicklungsprozesses verbessern.

Dieses Ergebnis steht im Einklang mit aktuellen Erkenntnissen aus der Forschung, die den Einfluss von Eingabequalität (Prompt und Requirement Clarity) auf die Modellleistung betonen (vgl. Wang et al., 2024; Guo et al., 2023). Besonders relevant ist, dass der beobachtete Effekt nicht auf die Größe oder Leistungsfähigkeit des Modells, sondern auf die Form und Struktur der Eingaben zurückzuführen ist – ein wichtiger Hinweis für die Gestaltung zukünftiger KI-gestützter Engineering-Workflows.

Hinsichtlich FF.2 zeigen die zehnfachen Wiederholungen je Projektszenario, dass die Simulationsumgebung eine hohe Reproduzierbarkeit aufweist. Die Standardabweichung der Erfolgsmetriken lag in beiden Szenarien unter 10 %, was auf eine stabile und deterministische Prozessausführung hinweist. Die Wahl einer niedrigen Modelltemperatur (0,2) erwies sich als geeignet, um stochastische Schwankungen zu minimieren, ohne die semantische Qualität der Antworten wesentlich zu beeinträchtigen.

Insgesamt belegen die Ergebnisse, dass die entwickelte Umgebung die definierten Ziele erfüllt:

* Sie erlaubt die kontrollierte Simulation von Software-Engineering-Prozessen in GitLab,
* sie reagiert sensibel auf Änderungen der Eingabequalität,
* und sie produziert bei gleicher Konfiguration reproduzierbare Ergebnisse.

Damit kann der entwickelte Prototyp als funktionsfähige Simulationsumgebung betrachtet werden, die das Potenzial besitzt, künftig in der empirischen KI-Forschung im Software Engineering als methodische Grundlage eingesetzt zu werden.

## Limitationen

Trotz der positiven Ergebnisse weist die durchgeführte Evaluation mehrere Einschränkungen auf, die bei der Interpretation berücksichtigt werden müssen.

Erstens ist die Stichprobengröße mit jeweils zehn Runs pro Projektszenario begrenzt. Zwar ermöglicht sie eine qualitative Bewertung, jedoch sind die Ergebnisse statistisch nur eingeschränkt generalisierbar. Eine höhere Anzahl von Durchläufen oder Variationen des Modells (z. B. mit anderen LLMs) wäre notwendig, um belastbare Signifikanzwerte zu erzielen.

Zweitens wurde ausschließlich ein Java-Maven-Projekt als technologische Basis verwendet. Andere Technologie-Stacks (z. B. Node.js, Python oder C#) könnten abweichende Ergebnisse erzeugen, da die Fehlertoleranz, Build-Dauer und Teststruktur differieren. Die Generalisierbarkeit der Ergebnisse auf andere Toolchains ist daher eingeschränkt.

Drittens unterliegt die Kommunikation mit den LLMs API-bedingten Schwankungen. Netzwerklatenzen, temporäre Modellupdates oder serverseitige Caching-Effekte können subtile Unterschiede in den Antworten hervorrufen. Obwohl die Temperatur konstant gehalten wurde, kann eine vollständige Deterministik nicht garantiert werden.

Viertens konzentrierte sich die Arbeit auf syntaktische Qualität der Anforderungen. Aspekte der semantischen Qualität (z. B. logische Konsistenz oder fachliche Vollständigkeit) wurden nicht explizit untersucht. Zukünftige Arbeiten sollten daher eine kombinierte Bewertung beider Dimensionen vornehmen.

Schließlich erfolgte die Evaluation in einer kontrollierten Simulationsumgebung mit einem begrenzten Agentenset. In realen Softwareprojekten wirken zusätzliche Faktoren wie Teamdynamik, Kontextwissen oder Domänenspezifika, die in dieser Arbeit nicht abgebildet werden konnten.

## Herausforderungen

Die Entwicklung und Ausführung der Simulationsumgebung brachte mehrere technische und konzeptionelle Herausforderungen mit sich.

### Technische Herausforderungen

Ein zentrales Risiko bestand im Betrieb des MCP-Servers (Model Context Protocol), der als Kommunikationsschnittstelle zwischen den Agenten, dem GitLab-System und den LLMs fungiert.

Die Stabilität des MCP-Servers ist entscheidend für den deterministischen Ablauf der Simulation. Zeitweise kam es zu Timeouts oder Thread-Kollisionen, wenn mehrere Agenten gleichzeitig Aktionen auslösten. Eine robuste Fehlerbehandlung und Wiederholungslogik waren daher erforderlich, um Unterbrechungen im Workflow zu vermeiden.

Auch das Context Engineering erwies sich als komplex: Jeder Agent benötigt einen klar abgegrenzten, aber ausreichend informativen Kontext, um effektiv zu agieren. Eine zu enge Kontextdefinition führt zu Informationsverlust, während eine zu weite Definition zu Halluzinationen oder redundanten Aktionen führen kann.

Das Prompt Engineering stellte eine weitere Herausforderung dar. Schon kleine Änderungen im Prompt-Layout oder in der Formulierung der Instruktionen beeinflussten das Verhalten der Agenten signifikant. Besonders auffällig war dies bei der Interaktion zwischen dem Coding- und Testing-Agent, wo unklare Prompts zu ineffizienten Testgenerierungen führten.

### Organisatorische und methodische Herausforderungen

Die Kombination aus Software-Engineering-Methodik (CI/CD) und KI-basiertem Agentenverhalten stellte hohe Anforderungen an die Versuchssteuerung.

Da klassische Metriken aus dem Software Engineering (z. B. Test-Coverage) mit KI-Metriken (z. B. Semantikqualität) kombiniert werden mussten, war eine sorgfältige Definition der Evaluationsmetriken notwendig, um valide Aussagen treffen zu können.

Zudem zeigte sich, dass Reproduzierbarkeit und Determinismus bei LLM-basierten Systemen nie vollständig garantiert werden können. Eine transparente Dokumentation aller Parameter und Seeds ist daher essenziell, um Experimente nachvollziehbar zu halten.

## Ausblick

Die vorliegende Arbeit hat gezeigt, dass eine automatisierte Simulationsumgebung für KI-basierte Softwareentwicklungsprozesse technisch realisierbar und wissenschaftlich nutzbar ist.

Gleichzeitig eröffnet sie eine Vielzahl von weiterführenden Forschungs- und Entwicklungsansätzen:

1. **Erweiterung der Agentenarchitektur:**

Der Einsatz zusätzlicher spezialisierter Agenten, etwa für Code-Review, Refactoring oder Architekturentscheidungen, könnte die Simulation realistischer und komplexer machen.

1. **Integration weiterer Programmiersprachen und Frameworks:**

Durch die Unterstützung anderer Technologien (z. B. Python, Rust, TypeScript) kann die Generalisierbarkeit der Ergebnisse verbessert und der Einfluss technischer Variablen untersucht werden.

1. **Adaptive Temperatursteuerung:**

Eine dynamische Anpassung der LLM-Temperatur in Abhängigkeit von der Aufgabenkomplexität könnte helfen, den Trade-off zwischen Kreativität und Reproduzierbarkeit besser zu steuern.

1. **Erweiterte Qualitätsmetriken:**

Neben syntaktischer könnten künftig auch semantische Anforderungsqualitäten (z. B. logische Konsistenz, Vollständigkeit, Ambiguität) automatisiert bewertet werden.

1. **Einsatz in der Forschungspraxis:**

Die entwickelte Umgebung kann künftig als Werkzeug zur systematischen Evaluation von LLMs im Software Engineering dienen – etwa zur Vergleichsstudie verschiedener Modelle oder Promptstrategien.

Langfristig eröffnet sich die Möglichkeit, die Simulationsumgebung zu einem offenen Benchmarking-Framework auszubauen, das reproduzierbare Experimente mit unterschiedlichen LLMs und Projektkonfigurationen ermöglicht. Damit könnte sie einen Beitrag zur Standardisierung und Objektivierung der LLM-Evaluation in Software-Engineering-Kontexten leisten.

# Fazit

# Anhang

**Eidesstattliche Versicherung**

Hiermit versichere ich, *Vorname/Name*, geb. am *XX.XX.XXXX*, Matrikel-Nr. *12345678*, an Eides statt, dass die an diese Versicherung angefügte schriftliche Ausarbeitung selbstständig und ohne jede unerlaubte Hilfe angefertigt wurde, dass sie noch keiner anderen Stelle zur Prüfung vorgelegen hat und dass sie weder ganz noch im Auszug veröffentlicht worden ist.

Die Stellen der Arbeit, die anderen Werken und Quellen (auch Internetquellen) dem Wortlaut oder dem Sinn nach entnommen sind, habe ich in jedem einzelnen Fall als Entlehnung mit exakter Quellenangabe kenntlich gemacht. Zudem versichere ich, dass ich keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt habe.

Mir ist bekannt, dass diese Arbeit auch auf elektronischem Wege auf Einhaltung wissenschaftlicher Standards überprüft wird und im Falle eines Plagiats als Täuschungsversuch bewertet werden kann.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ort, Datum Unterschrift

**Sperrvermerk *(i.d.R. nur bei Abschlussarbeiten)***

Die vorliegende Abschlussarbeit beinhaltet interne vertrauliche Informationen der Firma *XYZ*. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch in digitaler Form - gefertigt werden. Die Abschlussarbeit ist nur dem Studierenden-Servicebüro, den Korrektoren sowie den Mitgliedern des Prüfungsausschusses zugänglich zu machen. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma *XYZ*. Erlaubt ist das Einstellen von Textpassagen der Abschlussarbeit in digitaler Form zur Prüfung durch die Plagiatssoftware „URKUND“. Diese Sperrklausel gilt für einen Zeitraum von fünf Jahren.

References

[1] Ashish Vaswani *et al.,* "Attention is All you Need,"

[2] A. Nguyen-Duc, P. Abrahamsson, and F. Khomh, Eds., *Generative AI for effective software development*. Cham: Springer, 2024.

[3] S. K. Jabrw and Q. I. Sarhan, "A Systematic Survey on Large Language Models for Code Generation," *ARO*, vol. 13, no. 2, pp. 83–99, 2025, doi: 10.14500/aro.12159.

[4] X. Hou *et al.,* "Large Language Models for Software Engineering: A Systematic Literature Review," Aug. 2023.

[5] Z. Chen and L. Jiang, "Evaluating Software Development Agents: Patch Patterns, Code Quality, and Issue Complexity in Real-World GitHub Scenarios," Oct. 2024. [Online]. Available: http://​arxiv.org​/​pdf/​2410.12468v2

[6] L. Wang *et al.,* "A survey on large language model based autonomous agents," *Front. Comput. Sci.*, vol. 18, no. 6, 2024, doi: 10.1007/s11704-024-40231-1.

[7] L. Applis, Y. Zhang, S. Liang, N. Jiang, L. Tan, and A. Roychoudhury, "Unified Software Engineering agent as AI Software Engineer," Jun. 2025. [Online]. Available: http://​arxiv.org​/​pdf/​2506.14683v1

[8] Y. Wang *et al.,* "Agents in software engineering: survey, landscape, and vision," *Autom Softw Eng*, vol. 32, no. 2, 2025, doi: 10.1007/s10515-025-00544-2.

[9] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Singapore Singapore, 2016, pp. 426–437.

[10] L. Zhang *et al.,* "SWE-bench Goes Live!," May. 2025. [Online]. Available: http://​arxiv.org​/​pdf/​2505.23419v2

[11] Y. Chang *et al.,* "A Survey on Evaluation of Large Language Models," *ACM Trans. Intell. Syst. Technol.*, vol. 15, no. 3, pp. 1–45, 2024, doi: 10.1145/3641289.

[12] IEEE Computer Society/Software & Systems Engineering Standards Committee, "ISO/IEC/IEEE 29148:2018﻿,"

[13] A. Hemmat, M. Sharbaf, S. Kolahdouz-Rahimi, K. Lano, and S. Y. Tehrani, "Research directions for using LLM in software requirement engineering: a systematic review," *Front. Comput. Sci.*, vol. 7, 2025, doi: 10.3389/fcomp.2025.1519437.

[14] M. Yang, N. Lovett, B. Li, and Z. Hou, "Towards Dynamic Learner State: Orchestrating AI Agents and Workplace Performance via the Model Context Protocol," *Education sciences*, vol. 15, no. 8, p. 1004, 2025, doi: 10.3390/educsci15081004.

[15] Model Context Protocol, *What is the Model Context Protocol (MCP)? - Model Context Protocol* (accessed: Sep. 7 2025).

[16] S. Baltes *et al.,* "Guidelines for Empirical Studies in Software Engineering involving Large Language Models," Aug. 2025. [Online]. Available: http://​arxiv.org​/​pdf/​2508.15503v2

[17] A. Ferrari, P. Spoletini, and S. Gnesi, "Ambiguity and tacit knowledge in requirements elicitation interviews," *Requirements Eng*, vol. 21, no. 3, pp. 333–355, 2016, doi: 10.1007/s00766-016-0249-3.

[18] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007, doi: 10.2753/MIS0742-1222240302.

[19] *GitLab Docs.* [Online]. Available: https://​docs.gitlab.com​/​ (accessed: Sep. 8 2025).

[20] I.-C. Donca, O. P. Stan, M. Misaros, D. Gota, and L. Miclea, "Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects," *Sensors (Basel, Switzerland)*, vol. 22, no. 12, 2022, doi: 10.3390/s22124637.

[21] M. R. Wróbel, J. Szymukowicz, and P. Weichbroth, "Using Continuous Integration Techniques in Open Source Projects—An Exploratory Study," *IEEE Access*, vol. 11, pp. 113848–113863, 2023, doi: 10.1109/ACCESS.2023.3324536.

[22] *Merge requests | GitLab Docs.* [Online]. Available: https://​docs.gitlab.com​/​user/​project/​merge\_​requests/​ (accessed: Sep. 8 2025).

[23] C. Bird and T. Menzies, Eds., *The art and science of analyzing software data*. Amsterdam: Elsevier, 2015. [Online]. Available: http://​www.sciencedirect.com​/​science/​book/​9780124115194

[24] M. V. Mantyla and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?," *IIEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 430–448, 2009, doi: 10.1109/TSE.2008.71.

[25] Yangyang Zhao, "The Impact of Continuous Integration on Other Software Development Practices: A Large-Scale Empirical Study,"

[26] P. Rodríguez *et al.,* "Continuous deployment of software intensive products and services: A systematic mapping study," *Journal of Systems and Software*, vol. 123, pp. 263–291, 2017, doi: 10.1016/j.jss.2015.12.015.

[27] IEEE, "A Closer Look on the Difficulties to Determine the Quality of Software Requirements,"

[28] Artem Boyarchuk, "Approach to the Analysis of Software Requirements Specification on Its Structure Correctness,"

[29] I. Atoum *et al.,* "Challenges of Software Requirements Quality Assurance and Validation: A Systematic Literature Review," *IEEE Access*, vol. 9, pp. 137613–137634, 2021, doi: 10.1109/ACCESS.2021.3117989.

[30] L. Montgomery, D. Fucci, A. Bouraffa, L. Scholz, and W. Maalej, "Empirical research on requirements quality: a systematic mapping study," *Requirements Eng*, vol. 27, no. 2, pp. 183–209, 2022, doi: 10.1007/s00766-021-00367-z.

[31] E. Buchmann and S. Hauser, "Text Mining for Standardized Quality Criteria of Natural-Language IT-Requirements," in *2021 IEEE 29th International Requirements Engineering Conference (RE)*, Notre Dame, IN, USA, Sep. 2021 - Sep. 2021, pp. 390–395.

[32] C. Kumar Sahu, V. Rai, and V. Roshan, "Well-Formed Quality of System Requirements for Verifying to ISO 29148-2018: A Natural Language Processing (NLP) Based Framework and Quantitative Metric," in *Volume 2B: 44th Computers and Information in Engineering Conference (CIE)*, Washington, DC, USA, 08252024.

[33] S. Yao *et al.,* "Tree of Thoughts: Deliberate Problem Solving with Large Language Models," May. 2023. [Online]. Available: http://​arxiv.org​/​pdf/​2305.10601

[34] J. D. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why Johnny Can’t Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, Hamburg Germany, 2023, pp. 1–21.

[35] *Function calling - OpenAI API.* [Online]. Available: https://​platform.openai.com​/​docs/​guides/​function-​calling (accessed: Sep. 8 2025).

[36] M. A. Ferrag, N. Tihanyi, D. Hamouda, L. Maglaras, and M. Debbah, "From Prompt Injections to Protocol Exploits: Threats in LLM-Powered AI Agents Workflows," Jun. 2025. [Online]. Available: http://​arxiv.org​/​pdf/​2506.23260

[37] M. J. Wooldridge, *An introduction to multiagent systems*. Chichester, West Sussex: Wiley, 2009.

[38] N. R. Jennings, K. Sycara, and M. Wooldridge, "A Roadmap of Agent Research and Development," *Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, pp. 7–38, 1998, doi: 10.1023/A:1010090405266.

[39] S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, no. 27, pp. 79–80, 1995.

[40] W. Jager, R. Verbrugge, A. Flache, G. de Roo, L. Hoogduin, and C. Hemelrijk, *Advances in Social Simulation 2015*. Cham: Springer International Publishing, 2017.

[41] "THE LOGICAL MODELLING OF COMPUTATIONALMULTI-AGENT SYSTEMS,"

[42] R. Calegari, G. Ciatto, V. Mascardi, and A. Omicini, "Logic-based technologies for multi-agent systems: a systematic literature review," *Autonomous Agents and Multi-Agent Systems*, vol. 35, no. 1, 2021, doi: 10.1007/s10458-020-09478-3.

[43] T. Guo *et al.,* "Large Language Model based Multi-Agents: A Survey of Progress and Challenges," Jan. 2024. [Online]. Available: http://​arxiv.org​/​pdf/​2402.01680

[44] A. Plaat, M. van Duijn, N. van Stein, M. Preuss, P. van der Putten, and K. J. Batenburg, "Agentic Large Language Models, a survey," 2025.

[45] V. Botti, "Agentic AI and Multiagentic: Are We Reinventing the Wheel?," 2025.

[46] D. J. Moore, "A Taxonomy of Hierarchical Multi-Agent Systems: Design Patterns, Coordination Mechanisms, and Industrial Applications," 2025.