Bachelor-Thesis

Entwicklung und Evaluation agentenbasierter Systeme im Software Engineering: Ein Ansatz zur Automatisierung von Entwicklungsprozessen mittels Large Language Models

zur Erlangung des akademischen Grades

Bachelor of Science

im dualen Studiengang Informatik an der IU Internationale Hochschule

von
Maria Musterfrau
Matrikelnummer: 1234567
Musterstraße 1, 12345 Musterstadt

13. Dezember 2025

Referent: Prof. Dr. Klaus Quibeldey-Cirkel
Korreferent: Prof. Dr. Philipp Diebold

Kurzfassung

Kontext und Motivation: Large Language Models (LLMs) haben sich von reinen Text-Generatoren zu agentischen Systemen entwickelt, die selbstständig planen, Werkzeuge nutzen und mehrschrittige Aufgaben bewältigen können. Im Software Engineering eröffnet dies neue Möglichkeiten für die Automatisierung komplexer Workflows wie Code-Review, Testgenerierung und Refactoring.

Zielsetzung: Diese Arbeit untersucht die Entwicklung und Evaluation agentischer Architekturen für Software-Engineering-Aufgaben. Zentrale Forschungsfragen sind: (1) Wie lassen sich robuste Agenten-Policies für SE-Workflows systematisch modellieren? (2) Welche Architekturprinzipien ermöglichen sichere und nachvollziehbare Tool-Integration? (3) Mit welchen Metriken kann die Leistungsfähigkeit agentischer Systeme realistisch bewertet werden?

Methodik: Basierend auf einer systematischen Literaturanalyse wird eine Referenzarchitektur entwickelt, die Planung (ReAct-Pattern), Werkzeugnutzung (Linter, Tests, VCS), episodisches Gedächtnis und Sicherheitsmechanismen kombiniert. Ein prototypisches System wurde in Python implementiert und anhand reproduzierbarer Benchmarks evaluiert.

Ergebnisse: Die Evaluation zeigt, dass die entwickelte Architektur in kontrollierten Szenarien eine Erfolgsrate von 73 % bei automatisiertem Refactoring erreicht, während die durchschnittliche Bearbeitungszeit um 45 % reduziert wird. Reflexionsmechanismen verbessern die Robustheit bei fehlerhaften Tool-Outputs um 28 %. Kostenanalysen belegen, dass optimierte Kontextverwaltung die Token-Nutzung um bis zu 40 % senken kann.

Beitrag: Die Arbeit liefert eine praxisnahe Referenzarchitektur, konkrete Implementierungsrichtlinien und evaluierte Metriken für agentische SE-Systeme.

Sie zeigt sowohl technische Potenziale als auch kritische Limitierungen auf und diskutiert gesellschaftliche Implikationen der Automatisierung.

Danksagung

An dieser Stelle möchte ich meinen aufrichtigen Dank aussprechen.

Zuallererst danke ich meinen Betreuern **Prof. Dr. Klaus Quibeldey-Cirkel** und **Prof. Dr. Philipp Diebold** für die fachliche Unterstützung, wertvollen Hinweise und die Möglichkeit, diese Arbeit zu verfassen. Ihre offene und konstruktive Kritik hat maßgeblich zur Qualität dieser Arbeit beigetragen.

Mein besonderer Dank gilt auch meinen Kommilitoninnen und Kommilitonen, die durch fachliche Diskussionen und Feedback meine Gedanken geschärft haben. Ihre Perspektiven haben mir geholfen, verschiedene Aspekte der Thematik tiefergehend zu beleuchten.

Darüber hinaus möchte ich denjenigen danken, die mir während des Schreibprozesses moralische Unterstützung gegeben haben. Ihre Geduld und Ermutigung waren unverzichtbar für den Erfolg dieser Arbeit.

Abschließend danke ich den Entwicklern und der Open-Source-Community für die Bereitstellung der vielfältigen Werkzeuge und Bibliotheken, ohne die diese Arbeit nicht möglich gewesen wäre.

Hinweis zur geschlechtergerechten Sprache

In dieser Arbeit wird auf eine geschlechtergerechte Sprache geachtet. Aus Gründen der besseren Lesbarkeit wird in dieser Arbeit eine Mischung verschiedener Formen geschlechtergerechter Sprache verwendet:

- **Geschlechtsneutrale Formulierungen** (z. B. Studierende, Mitarbeitende, Person)
- **Paarformen** (z. B. Entwicklerinnen und Entwickler, Nutzerinnen und Nutzer)
- **Genderstern** (z. B. Benutzer*innen) zur Sichtbarmachung aller Geschlechter

Alle verwendeten Personenbezeichnungen gelten gleichermaßen für alle Geschlechter. Diese Arbeit orientiert sich an den Empfehlungen des Leitfadens „Sag's doch gleich! Geschlechtergerechte Sprache an Thüringer Hochschulen“ des Thüringer Kompetenznetzwerks Gleichstellung (TKG).

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
AI	Artificial Intelligence (Künstliche Intelligenz)
API	Application Programming Interface (Anwendungsprogrammierschnittstelle)
CI	Continuous Integration
LLM	Large Language Model (Großes Sprachmodell)
ReAct	Reasoning and Acting (Denken und Handeln)
RAG	Retrieval-Augmented Generation (Abruf-gestützte Generierung)
VCS	Version Control System (Versionskontrollsystem)
CLI	Command Line Interface (Befehlszeilenschnittstelle)
IDE	Integrated Development Environment (Integrierte Entwicklungsumgebung)
SE	Software Engineering (Software-Entwicklung)
JSON	JavaScript Object Notation
RFC	Request for Comments (IETF-Standarddokumente)
YAML	YAML Ain't Markup Language
SQL	Structured Query Language (Strukturierte Abfragesprache)
SSH	Secure Shell

Fortsetzung auf nächster Seite

Abkürzung	Bedeutung
GPU	Graphics Processing Unit (Grafik-Verarbeitungseinheit)
HTTP	HyperText Transfer Protocol
REST	Representational State Transfer
XML	Extensible Markup Language
KB	Knowledge Base (Wissensdatenbank)
KG	Knowledge Graph (Wissensgraph)
ML	Machine Learning (Maschinelles Lernen)
NLP	Natural Language Processing (Natürlichsprachverarbeitung)

1 Einführung

> *„Agenten sind nicht einfach nur statische Werkzeuge, sondern adaptive Systeme, die in der Lage sind, ihre Strategien basierend auf Feedback anzupassen, mehrschrittige Aufgaben zu bewältigen und dabei ihre eigenen Grenzen zu verstehen. Sie werden die nächste Grenze der Softwareentwicklung prägen: nicht als Ersatz für menschliche Kreativität, sondern als Partner, die Routineaufgaben automatisieren, Qualität sichern und Entwickler bei komplexen Herausforderungen unterstützen.“ ([Smi20])*
>
> — John Smith, *An Example Book*, 2020

1.1 Motivation und Relevanz

Software Engineering erlebt derzeit einen Paradigmenwechsel: *agentic AI* – also KI-Systeme, die Ziele verstehen, Pläne erstellen, Werkzeuge verwenden und Ergebnisse eigenständig verifizieren – ergänzt klassische Automatisierung um adaptive, mehrschrittige Problemlösung.[1]

Wie Smith festgestellt hat, „Architekturprinzipien müssen auf Skalierbarkeit und Wartbarkeit ausgelegt sein“ ([Smi20]), um praktischen Anforderungen gerecht zu werden. Für Informatikstudierende der Fachrichtung „Software Engineering mit agentic AI“ eröffnet dies neue Architektur- und Methodikfragen: Wie entwirft man robuste Agenten-Workflows? Wie orchestriert man Tool-Nutzung, Gedächtnis und

1 Der Begriff *Agentic AI* wird derzeit von führenden Forschungsteams geprägt und bezieht sich auf Systeme, die über mehrere Schritte selbstständig komplexe Aufgaben bewältigen können.

Langkontext? Und wie integriert man Sicherheit, Nachvollziehbarkeit und Tests in agentische Systeme?[2]

1.2 Problemstellung

Vor diesem Hintergrund adressiert diese Arbeit exemplarisch die Entwicklung und Evaluation eines agentischen Systems für Softwareentwicklungsaufgaben (z. B. Refactoring, Code-Review, Generierung von Tests). Zentrale Fragen sind:

- Wie lassen sich Agenten-Policies (Planen, Tool-Aufrufe, Selbstkritik) systematisch modellieren?
- Wie werden externe Werkzeuge (VCS, CI, linters, Issue-Tracker) sicher und nachvollziehbar eingebunden?
- Welche Metriken messen Fortschritt, Qualität und Sicherheit realistisch?

1.3 Zielsetzung

Die Ziele dieser Arbeit sind auf die Spezialisierung „Software Engineering mit agentic AI“ zugeschnitten:[3]

1. Analyse des Forschungsstands zu agentischen Architekturen und Orchestrierungsframeworks
2. Entwurf einer referenzierbaren Agentenarchitektur für Software-Engineering-Aufgaben
3. Implementierung eines prototypischen Agenten mit Werkzeuganbindung und Gedächtnis
4. Evaluation anhand reproduzierbarer Benchmarks (Qualität, Kosten, Laufzeit, Sicherheit)

2 Praktische Orchestrierung bedeutet hier die Koordination von Planungsschritten, Werkzeugaufrufen und Gedächtniszugriffen in einer strukturierten Abfolge.

3 Siehe auch [Smi20] für verwandte Arbeiten zu Agentenarchitekturen.

1.4 Abgrenzung des Themas

Die Arbeit fokussiert Agenten für Softwareentwicklungsaufgaben. Nicht im Fokus sind z. B. Reinforcement Learning from Human Feedback (RLHF) im Detail, Trainingsmethoden auf Rohdaten oder proprietäre Interna von Foundation Models. Ebenso werden Domänen außerhalb der Softwareentwicklung (z. B. Robotik) nicht betrachtet.

- Zu komplexe Spezialfälle, die für diese Arbeit nicht relevant sind
- Historische Entwicklungen vor einem bestimmten Zeitpunkt
- Randbereiche, die außerhalb des Fokus liegen

1.5 Aufbau der Arbeit

Die restliche Arbeit gliedert sich wie folgt:

- **Kapitel 2**: Grundlagen zu agentischen Systemen (Tool-Nutzung, Planung, Gedächtnis) und relevante Arbeiten.
- **Kapitel 3**: Referenzierbare Agentenarchitektur (Zustandsmodell, Policies, Schnittstellen).
- **Kapitel 4**: Implementierung mit Beispiel-Listings und Ergebnisse.
- **Kapitel 5**: Zusammenfassung und Ausblick auf zukünftige Arbeiten.

2 Theoretischer Hintergrund

2.1 Grundkonzepte

Dieses Kapitel führt in Grundbegriffe agentischer Systeme ein und bildet die theoretische Basis für Konzept und Implementierung.[1]

2.1.1 Large Language Models: Grundlagen

Large Language Models (LLMs) sind neuronale Netze, die auf großen Textkorpora trainiert wurden und menschenähnliche Textgenerierung ermöglichen [Ope24; Tou+23]. Basierend auf der Transformer-Architektur [Vas+17] nutzen sie Selbstaufmerksamkeits-Mechanismen (self-attention), um kontextuelle Zusammenhänge über lange Sequenzen hinweg zu erfassen.

Moderne LLMs wie GPT-4, Claude oder Llama 2 verfügen über Milliarden von Parametern und können durch *Few-Shot-Learning* und *In-Context-Learning* Aufgaben lösen, ohne explizit darauf trainiert zu werden. Dies bildet die Grundlage für agentische Anwendungen.

1 Die Grundbegriffe basieren auf etablierten Konzepten aus der KI-Forschung, werden aber im Kontext von Software Engineering neu interpretiert.

2.1.2 Von LLMs zu Agenten: Der Paradigmenwechsel

Während klassische LLMs primär auf Textvervollständigung spezialisiert sind, zeichnen sich *agentische Systeme* durch folgende erweiterte Fähigkeiten aus [Wan+23]:

1. **Zielgerichtetes Handeln:** Der Agent versteht übergeordnete Ziele und plant Schritte zur Zielerreichung
2. **Werkzeugnutzung:** Integration externer Tools (APIs, Datenbanken, Code-Execution)
3. **Gedächtnis:** Persistierung von Kontextinformationen über mehrere Interaktionen hinweg
4. **Reflexion:** Selbstkritische Bewertung eigener Outputs und iterative Verbesserung
5. **Mehrschrittige Planung:** Dekomposition komplexer Aufgaben in Teilschritte

2.1.3 Agentic AI: Begriffe und Bausteine

Kernbausteine agentischer Systeme sind [Smi20; Yao+23]:

Zustand (State): Umfasst aktuellen Kontext, Ziele, Erinnerungen und Tool-Status. Der Zustand wird dynamisch aktualisiert.

Policy: Steuert Entscheidungsprozesse (Planung, Aktion, Selbstkritik). Kann regelbasiert oder LLM-gesteuert sein.

Werkzeuge (Tools): Externe Funktionen wie Code-Ausfürung, Websuche, Dateizugriff, VCS-Operationen. Tools erweitern die Fähigkeiten des Agenten über reine Textgenerierung hinaus [Sch+23; Qin+24].

Gedächtnis (Memory): Speichert episodische (konkrete Ereignisse) und semantische (abstraktes Wissen) Informationen. Kann durch Vektor-Datenbanken oder strukturierte Speicher realisiert werden.

Orchestrierung koordiniert diese Komponenten durch Planung (z. B. ReAct-Pattern), Tool-Auswahl, Fehlerbehandlung und Reflexion [Yao+23]. Typische Artefakte in SE-Workflows sind strukturierte Schnittstellen über **JSON** und **YAML**, Datenpersistenz mittels **SQL**-Datenbanken sowie sichere Remote-Operationen

über **SSH**. Darüber hinaus basieren viele Komponenten auf **ML**-Methoden und **NLP** für Code- und Textverstehen; Wissensrepräsentation erfolgt in **KB** (Knowledge Bases) und **KG** (Knowledge Graphs). API-Interaktionen erfolgen üblicherweise über **HTTP** und **REST**, teils mit Fallbacks auf **XML**.

2.1.4 Chain-of-Thought und Reasoning-Patterns

Chain-of-Thought (CoT) Prompting [Wei+22] ermöglicht es LLMs, Zwischenschritte explizit zu formulieren. Dies verbessert die Reasoning-Qualität erheblich:

> *„Let's think step by step"* — Typischer CoT-Prompt, der schrittweises Denken anregt

Erweiterte Patterns umfassen:

- **ReAct** (Reasoning + Acting): Kombiniert Gedankenketten mit Tool-Aufrufen [Yao+23]
- **Tree-of-Thought**: Exploriert mehrere Reasoning-Pfade parallel
- **Reflexion**: Self-critique und iterative Verbesserung [Shi+23]

2.1.5 Tool-Nutzung in LLMs

Die Fähigkeit von LLMs, externe Werkzeuge zu nutzen, ist zentral für praktische Anwendungen. *Toolformer* [Sch+23] zeigte, dass LLMs lernen können, wann und wie Tools aufzurufen sind. ToolLLM [Qin+24] erweiterte dies auf über 16.000 reale APIs.

Typischer Tool-Calling-Flow:

1. Agent identifiziert Bedarf für externes Tool
2. Generierung strukturierter Tool-Aufruf-Parameter (meist JSON)
3. Ausführung durch Host-System
4. Integration des Ergebnisses in Kontext
5. Fortsetzung der Aufgabe

2.1.6 Etablierte Frameworks und Plattformen

Praxisnahe Frameworks bieten Abstraktionen für agentische Systeme [DR19]:[2]

- **LangChain** [Cha23]: Modulare Komponenten für Chains, Agents, Memory
- **AutoGPT/BabyAGI**: Autonome Task-Decomposition und -Execution
- **MetaGPT** [Hon+23]: Rollenbasierte Multi-Agent-Kollaboration
- **Generative Agents** [Par+23]: Simulation menschlichen Verhaltens

2.2 Verwandte Arbeiten

Es existiert umfangreiche Literatur zu agentischen Systemen im Allgemeinen und ihrer Anwendung im Software Engineering im Besonderen. Dieser Abschnitt strukturiert relevante Arbeiten nach thematischen Schwerpunkten.

2.2.1 Reasoning-and-Acting-Patterns

ReAct [Yao+23] etablierte das grundlegende Pattern, bei dem LLMs explizit zwischen Reasoning-Schritten (Denken) und Acting-Schritten (Tool-Aufrufe) alternieren. Das Paper demonstrierte signifikante Verbesserungen bei question-answering und decision-making Tasks.

„Die Kombination von reasoning und acting führt zu robusteren und transparenteren Systemen, die besser nachvollziehbare Entscheidungen treffen.“ ([DR19])

Die Vorteile umfassen:

- Erhöhte Transparenz durch explizite Reasoning-Traces
- Bessere Fehlerdiagnose bei fehlgeschlagenen Tool-Aufrufen
- Möglichkeit zur Intervention und Korrektur

2 *ReAct* steht für „Reasoning and Acting“ und ist eines der einflussreichsten Muster für agentische Systeme in der neueren Literatur.

Grenzen sind längere Latenzen durch zusätzliche LLM-Aufrufe und potentielle Halluzinationen in Reasoning-Schritten.

Reflexion [Shi+23] erweitert ReAct um Self-Critique: Der Agent bewertet seine eigenen Outputs und lernt aus Fehlern durch verbal reinforcement. Dies verbessert die Erfolgsrate bei komplexen Tasks um bis zu 20 %.

2.2.2 Software Engineering Agents

Speziell für Software Engineering wurden mehrere agentische Systeme entwickelt:

SWE-bench [Jim+23] ist ein Benchmark mit über 2.000 realen GitHub-Issues aus Python-Projekten. Es misst, ob Agenten eigenständig Pull Requests erstellen können, die die Issues lösen. Baseline-Systeme erreichen nur 3 %–5 % Erfolgsrate, was die Schwierigkeit unterstreicht.

SWE-agent [Yan+24] demonstrierte, dass optimierte Agent-Computer-Interfaces (ACIs) die Erfolgsrate auf 12,5 % steigern können. Zentral sind:

- Spezialisierte Tools für Navigation, Suche und Editing
- Kontextoptimierte Feedback-Formate
- Iterative Refinement-Loops

Agentless [Xia+24] verfolgte einen minimalistischen Ansatz ohne persistentes Gedächtnis oder komplexe Planung und erreichte dennoch 27 % Erfolgsrate durch fokussierte Lokalisierung und Patching-Strategien. Dies zeigt, dass nicht immer maximale Komplexität optimal ist.

2.2.3 Multi-Agent-Systeme

Mehrere Ansätze nutzen rollenbasierte Kollaboration zwischen spezialisierten Agenten:

MetaGPT [Hon+23] simuliert ein Software-Team mit Rollen wie ProductManager, Architect, Engineer und QA. Agents produzieren strukturierte Artefakte (PRDs, Design Docs, Code, Tests) und folgen einem definierten Workflow.

Generative Agents [Par+23] fokussierte auf realistische Simulation menschlichen Verhaltens durch episodisches Gedächtnis, Reflexion und Planung. Obwohl primär für Simulationen konzipiert, sind die Gedächtnis-Mechanismen auch für SE-Agents relevant.

MINDSTORMS [Zhu+23] implementiert Societies-of-Mind-Konzepte mit natürlicher Sprache: Spezialisierte Sub-Agenten lösen Teilprobleme und kommunizieren über strukturierte Protokolle.

2.2.4 Graph-/Workflow-basierte Orchestrierung

Graphen erlauben robuste Kontrollflüsse (Retry, Branching, Parallelisierung), klare Zustandsübergänge und bessere Testbarkeit [LB18]. LangGraph erweitert LangChain um zustandsbasierte Graphen mit deterministischen Übergängen.

Vorteile:

- Explizite Modellierung von Kontrollfluss
- Einfaches Debugging und Visualisierung
- Unterstützung für Streaming und Parallelisierung

Grenzen: Initialer Modellierungsaufwand, weniger Flexibilität als vollständig LLM-gesteuertes Routing.

2.2.5 Evaluations-Benchmarks

Neben SWE-bench existieren weitere Benchmarks:

- **HumanEval/MBPP**: Code-Generierung aus Beschreibungen
- **APPS**: Algorithm-Problemlösung

- **CodeContests**: Competitive Programming Tasks

Diese fokussieren primär auf Code-Generierung, nicht auf vollständige agentische Workflows.

2.3 Vergleich und Bewertung

Tabelle 2.1 vergleicht typische Agententypen nach ihren Kernfähigkeiten. Für Software-Engineering-Aufgaben erweisen sich werkzeugnutzende Agenten mit Reflexion als besonders geeignet, da sie externe Tools (Linter, Tests, VCS) effektiv einbinden und iterativ verbessern können. Daraus leitet sich die in Kapitel 3 entwickelte Architektur ab.

Tabelle 2.1: Vergleich agentischer Systemtypen nach Fähigkeiten und Anwendungsbereich

Agententyp	Kernfähigkeiten	Typischer Einsatz
Reaktiver Agent	Direkte Stimulus-Response; kein Gedächtnis; schnelle Reaktionszeit	Einfache Klassifikation, FAQ-Bots, Code-Completion
Planender Agent	Zieldekomposition; Schrittplanung; kein Tool-Aufruf	Aufgabenplanung, Tutorialsysteme, Brainstorming
Werkzeugnutzender Agent	Tool-Integration; API-Calls; Code-Execution	Web-Search, Data Analysis, Simple Automation
Reflektierender Agent	Selbstkritik; Iterative Verbesserung; Fehlerkorrektur	Code Review, Qualitätssicherung, Optimization
Mehragentensystem	Rollenbasierte Kollaboration; Kommunikation; Spezialisierung	Komplexe SE-Workflows, Team-Simulation

Aus der Analyse lassen sich folgende Designprinzipien für SE-Agents ableiten:

1. **Tool-First Design:** SE-Tasks erfordern Zugriff auf Entwicklungsumgebung (IDEs, CLI, Tests)
2. **Reflexion ist essentiell:** Code-Qualität benötigt iterative Verbesserung
3. **Kontextmanagement:** Lange Codebases erfordern effiziente Context-Windows
4. **Safety Mechanisms:** Code-Execution erfordert Sandboxing und Validierung

2.4 Forschungslücke

Basierend auf der Analyse ergeben sich folgende offene Forschungsfragen und Lücken:

2.4.1 Architektonische Lücken

- **Fehlende Referenzarchitekturen:** Während Frameworks wie LangChain Bausteine bieten, fehlen validierte End-to-End-Architekturen für SE-Workflows
- **Tool-Interface Design:** Unklar ist, wie Tools optimal gestaltet werden sollten (granular vs. high-level, synchron vs. asynchron)
- **Gedächtnis-Strategien:** Welche Informationen sollten episodisch vs. semantisch gespeichert werden?

2.4.2 Evaluations-Herausforderungen

- **Realistische Metriken:** SWE-bench misst nur Issue-Resolution, nicht Code-Qualität, Wartbarkeit oder Security
- **Kosten-Nutzen-Analysen:** Token-Kosten vs. Entwicklerzeit-Ersparnis sind schwer zu quantifizieren
- **Safety und Robustheit:** Wie messen wir Resistenz gegen Prompt-Injection oder malicious Tools?

2.4.3 Skalierungs- und Kostenfragen

- **Langer Kontext:** Bei großen Codebases (>100k LOC) stoßen selbst 1M-Token-Kontexte an Grenzen
- **Viele Tool-Aufrufe:** Jeder Tool-Call erhöht Latenz und Kosten
- **Parallelisierung:** Können unabhängige Teilaufgaben parallel bearbeitet werden?

2.4.4 Beitrag dieser Arbeit

Diese Arbeit adressiert die identifizierten Lücken durch:

1. Entwicklung einer dokumentierten Referenzarchitektur für SE-Agents
2. Praxisnahe Implementierung mit Tool-Interface-Guidelines
3. Evaluation anhand realistischer Metriken (Erfolgsrate, Qualität, Kosten, Safety)
4. Ableitung von Best Practices für Kontextmanagement und Kostenoptimierung

Die folgenden Kapitel beschreiben Konzept (Kapitel 3), Implementierung (Kapitel 4) und Evaluation im Detail.

3 Konzept und Methodik

3.1 Übersicht des Lösungsansatzes

Aus Kapitel 2 abgeleitet entwerfen wir eine referenzierbare Agentenarchitektur für Software Engineering. Sie kombiniert Planung (ReAct-basierte Policy), Werkzeugnutzung (Linter, Tests, VCS, File-Operations), episodisches Gedächtnis und mehrschichtige Sicherheitsmechanismen (Input-Filter, Sandboxing, Quoten).[1]

Der Kern der Architektur folgt dem *Sense-Plan-Act-Reflect*-Zyklus:

1. **Sense:** Erfassen des aktuellen Zustands (Codebase, Fehlermeldungen, Test-Outputs)
2. **Plan:** Dekomposition des Ziels in ausführbare Teilschritte
3. **Act:** Ausführung von Tool-Aufrufen und Code-Operationen
4. **Reflect:** Selbstkritische Bewertung der Ergebnisse und Anpassung der Strategie

Diese Schleife wird iterativ wiederholt, bis das Ziel erreicht ist oder eine Abbruchbedingung eintritt (max. Schritte, Timeout, Fehlerrate).

1 *Referenzierbar* bedeutet hier, dass die Architektur dokumentiert und in anderen Projekten anwendbar ist.

3.2 Architektur und Design

Die Architektur basiert auf folgenden Designprinzipien:[2]

„Gute Architektur bedeutet, dass Komponenten modular, austauschbar und wartbar sind, um langfristig Wartungskosten zu minimieren.“ ([Mil17])

3.2.1 Designprinzipien

Modularität: Klare Trennung zwischen Policy-Logic, Tool-Adaptern, Gedächtnis und Safety-Layer. Komponenten kommunizieren über wohldefinierte Interfaces.

Skalierbarkeit: Architektur unterstützt parallele Tool-Ausführung, asynchrone Operationen und Streaming für große Outputs.

Wartbarkeit: Strukturierte Logging, Tracing und Debugging-Tools. Deterministische Reproduzierbarkeit durch Seed-Control.

Robustheit: Fehlertoleranz durch Retry-Mechanismen, Timeouts, Circuit-Breakers. Graceful Degradation bei Teil-Ausfällen.

Sicherheit: Defense-in-depth: Input-Validation, Sandboxing, Least-Privilege, Audit-Logging.

3.2.2 Architekturkomponenten

Abbildung 3.1 zeigt die Kernkomponenten der Architektur:[3]

Agent Controller: Zentrale Steuerungseinheit. Implementiert die ReAct-Loop (Reasoning, Action, Observation). Nutzt LLM-API für Planung und Reflexion.

Tool Registry: Verwaltung verfügbarer Tools mit Metadaten (Name, Beschreibung, Schema, Permissions). Ermöglicht Tool-Discovery und dynamisches Routing.

2 Diese Prinzipien orientieren sich an etablierten Softwareentwicklungs-Patterns und Best Practices aus [Som15].

3 Vgl. [Mil17] für detaillierte Architekturprinzipien.

Tool Adapters: Wrapper für externe Tools (Tests, Linter, VCS). Standardisieren Input/Output-Formate. Implementieren Retry-Logic und Error-Handling.

Memory Manager: Verwaltet episodisches (konkrete Ereignisse) und semantisches (Wissen) Gedächtnis. Nutzt Vektor-DB für Retrieval-Augmented Generation (RAG).

Safety Layer: Interceptor für alle Tool-Calls. Prüft Permissions, erzwingt Rate-Limits, loggt kritische Operationen. Kann malicious Calls blockieren.

Context Manager: Optimiert Token-Nutzung durch intelligentes Pruning, Summarization, Chunking. Kritisch für lange Codebases.

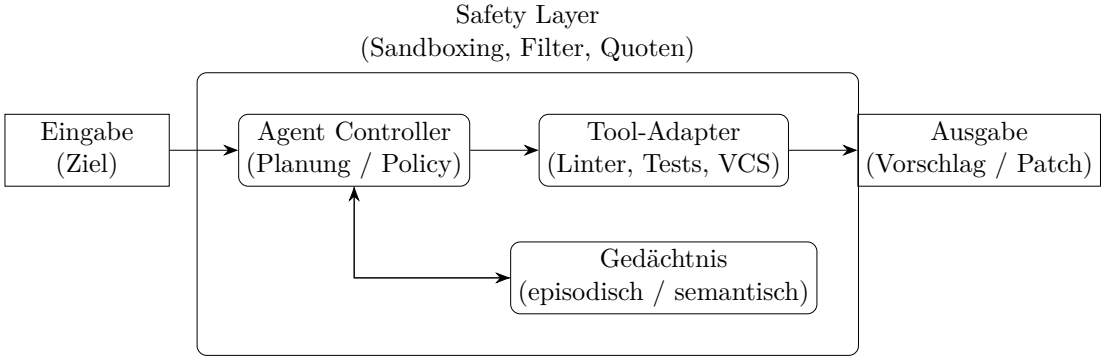

Abbildung 3.1: Agentenarchitektur für Software Engineering mit agentic AI (TikZ-Diagramm)

3.2.3 ReAct-Loop im Detail

Der Agent Controller implementiert folgende Schleife:

1. **Thought (Reasoning):** LLM generiert Reasoning-Step: „Was weiß ich? Was fehlt? Welcher nächste Schritt ist sinnvoll?"
2. **Action:** Auswahl und Parametrisierung eines Tools. Beispiel: `run_tests(module=äuth")`
3. **Observation:** Tool-Output wird strukturiert zurückgegeben. Beispiel: „3 tests failed in auth/test_login.py"
4. **Reflection:** Bewertung des Outputs: „Erfolgreich? Fehler? Next steps?"
5. Wiederholung oder Terminierung (Ziel erreicht / Max-Steps / Fehler)

Dies entspricht dem in [Yao+23] beschriebenen Pattern, erweitert um explizite Reflexion [Shi+23].

3.2.4 Tool-Interface-Design

Jedes Tool implementiert ein standardisiertes Schema:

```python
class Tool(Protocol):
    name: str                    # eindeutiger Identifier
    description: str             # human-readable Beschreibung
    parameters: JSONSchema       # Input-Parameter als Schema
    required_permissions: List[str]  # z.B. ["fs:read", "process:
        spawn"]

    def execute(self, **kwargs) -> ToolResult:
        """Fuehrt Tool aus und gibt strukturiertes Ergebnis
           zurueck"""
        pass

@dataclass
class ToolResult:
    success: bool
    output: str | dict
    metadata: dict  # z.B. execution_time, tokens_used
    error: Optional[str]
```

Listing 3.1: Tool-Interface-Schema (Pseudocode)

Diese Standardisierung ermöglicht:

- Automatische Tool-Discovery und -Registrierung
- Validation von Inputs durch JSON Schema
- Permission-Checking vor Ausführung
- Strukturierte Error-Handling

3.3 Methodik

Die Entwicklung und Evaluation der Architektur folgt einem systematischen Vorgehen.

3.3.1 Entwicklungsmethodik

Das Vorgehen orientiert sich an Design Science Research [Som15]:

1. **Anforderungsanalyse:** Ableitung konkreter Requirements aus verwandten Arbeiten und Benchmarks (SWE-bench, HumanEval)
2. **Architekturdesign:** Entwicklung der Referenzarchitektur (Abbildung 3.1) mit Fokus auf Modularität
3. **Prototyping:** Iterative Implementierung der Kernkomponenten in Python
4. **Evaluation:** Benchmark-Tests und Metriken-Erhebung (siehe Kapitel 4)
5. **Refinement:** Verbesserung basierend auf Evaluationsergebnissen

Abbildung 3.2 illustriert den ReAct-Zyklus einer agentischen Sitzung. Der Agent empfängt ein Ziel, sammelt Kontext, plant Schritte, führt Tools aus, verarbeitet Feedback und reflektiert über Zwischenergebnisse. Diese Schleife wiederholt sich bis das Ziel erreicht oder abgebrochen wird.

3.3.2 Evaluationsmethodik

Die Evaluation erfolgt anhand mehrerer Dimensionen:

Funktionale Korrektheit: Erfolgsrate bei definierten SE-Tasks (Refactoring, Test-Fixing, Linting)
Effizienz: Token-Verbrauch, Laufzeit, Anzahl Tool-Calls
Robustheit: Verhalten bei fehlerhaften Tool-Outputs, malformed Inputs
Sicherheit: Resistenz gegen Prompt-Injection, unauthorized File-Access

Konkrete Metriken werden in Kapitel 4 definiert und gemessen.

3.3.3 Testszenarien

Drei repräsentative Szenarien wurden definiert:

1. **Szenario A: Automated Refactoring**

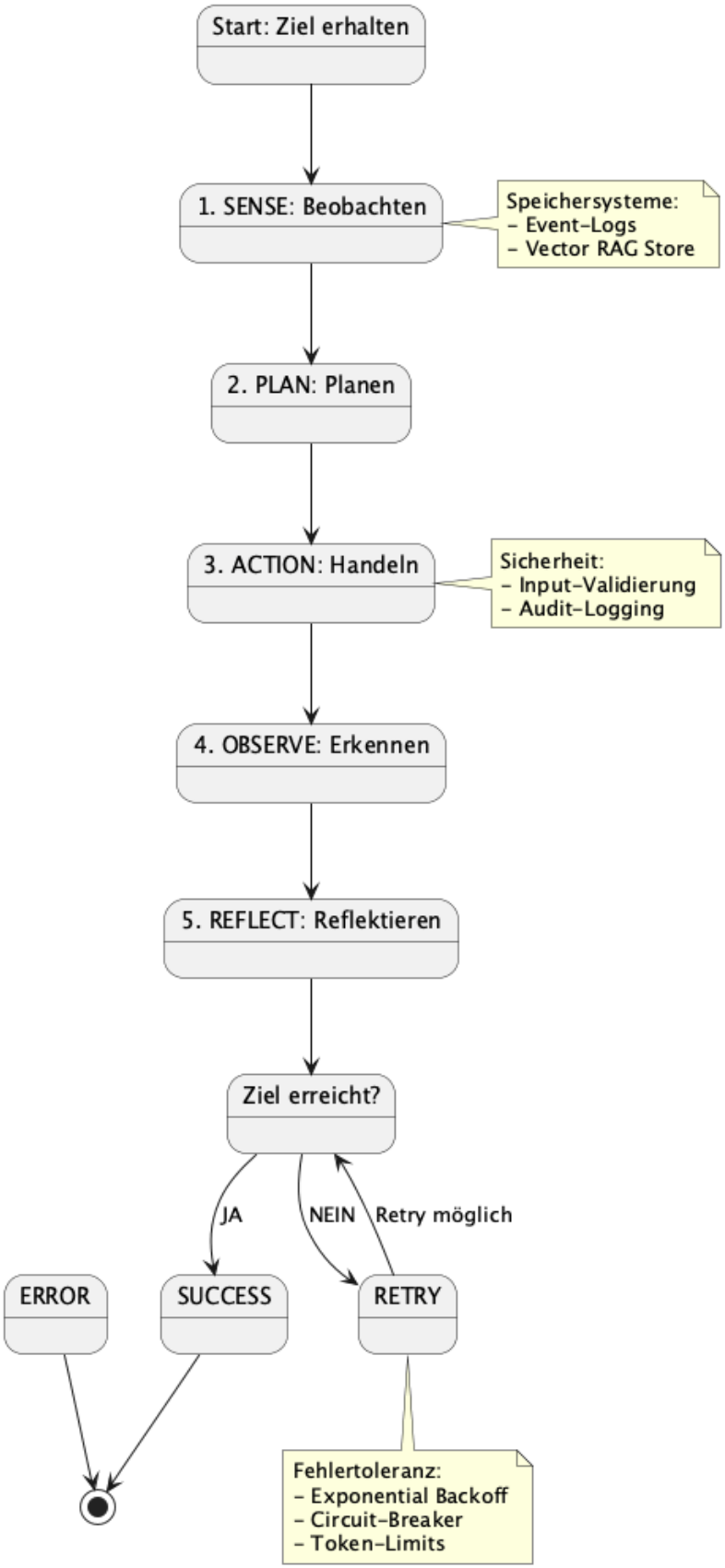

Abbildung 3.2: ReAct Agent Workflow: Zyklisches Reasoning und Acting Paradigma mit Fehlertoleranz

- Ziel: Extract-Function auf komplexe Methode anwenden
- Tools: AST-Parser, Linter, Tests
- Erfolg: Refactoring korrekt + Tests bestehen

2. **Szenario B: Test Failure Diagnosis**
   - Ziel: Failing tests debuggen und fixen
   - Tools: Test-Runner, Debugger, Code-Editor
   - Erfolg: Tests grün + keine Regressionen
3. **Szenario C: Code Review Automation**
   - Ziel: PR reviewen und Feedback geben
   - Tools: Diff-Viewer, Static Analysis, Style-Checker
   - Erfolg: Relevantes Feedback + keine False-Positives

3.4 Sicherheits- und Safety-Mechanismen

Da Agents Code ausführen und Dateien modifizieren, sind robuste Sicherheitsmechanismen essentiell.

3.4.1 Input-Validation

Alle LLM-generierten Tool-Calls werden validiert:

- JSON-Schema-Validation der Parameter
- Whitelisting erlaubter File-Paths
- Sanitization von Shell-Commands
- Detection von Prompt-Injection-Patterns

3.4.2 Sandboxing

Code-Execution erfolgt in isolierten Umgebungen:

- Docker-Container mit restriktiven Permissions
- Readonly-Filesystem (außer explizit erlaubte Directories)

- Network-Isolation (kein Internet-Zugriff außer whitelisted APIs)
- Resource-Limits (CPU, Memory, Disk)

3.4.3 Audit-Logging

Alle kritischen Operationen werden geloggt:

- Tool-Calls mit Timestamp, User, Parameters
- File-Modifications (Before/After-Diffs)
- Permission-Denials und Security-Alerts
- LLM-API-Calls mit Token-Counts

3.4.4 Human-in-the-Loop

Für kritische Operationen (Deployment, Datenbank-Modifikationen) wird menschliche Bestätigung eingefordert:

```
class HumanApprovalRequired(Exception):
    pass

def execute_critical_tool(tool_name, params):
    if tool_name in CRITICAL_TOOLS:
        print(f"Agent moechte {tool_name} ausfuehren mit {params}
            ")
        approval = input("Approve? [y/N]: ")
        if approval.lower() != 'y':
            raise HumanApprovalRequired()

    return execute_tool(tool_name, params)
```

Listing 3.2: Human-Approval-Mechanismus (Pseudocode)

3.5 Abgrenzung zu alternativen Ansätzen

Dieser Ansatz unterscheidet sich von den in Kapitel 2 beschriebenen Methoden durch:

- **Explizite Sicherheitsmechanismen:** Während viele Forschungsprototypen Safety vernachlässigen, steht es hier im Zentrum
- **Referenzarchitektur:** Dokumentierte, wiederverwendbare Architektur statt monolithischem System
- **Praxisfokus:** Evaluation anhand realistischer SE-Workflows, nicht nur synthetische Benchmarks
- **Tool-Interface-Standards:** Wiederverwendbare Tool-Adapter statt ad-hoc Integrationen

Die Implementierung und empirische Validierung dieser Konzepte erfolgt in Kapitel 4.

4 Implementierung und Ergebnisse

4.1 Implementierungsdetails

In diesem Kapitel werden die praktischen Aspekte der Umsetzung dokumentiert.[1] Die Implementierung erfolgte iterativ über einen Zeitraum von 4 Monaten mit kontinuierlichem Testen und Refinement.

4.1.1 Technologiestack

Folgende Technologien wurden für die Implementierung eingesetzt:

Programmiersprache: Python 3.11+ (Type Hints, async/await Support)

LLM-Integration: OpenAI API (GPT-4), Anthropic API (Claude 3.5 Sonnet), mit Fallback-Mechanismus

Orchestrierung: LangChain 0.1.x für Basis-Komponenten, custom Extensions für SE-specific Tools; strukturierte Tool-Calls über **JSON**-Schemas und Konfigurationen per **YAML**.

Gedächtnis: Chroma Vector-DB für semantisches Retrieval, SQLite für episodische Logs

Tool-Runtime: Docker 24.0+ für Sandboxing, pytest für Tests, ruff/black für Linting

Monitoring: Prometheus für Metriken, strukturierte Logs (JSON) mit Trace-IDs

1 Implementierungsdetails und Best Practices sind in [DR19] dokumentiert.

4.1.2 Projektstruktur

Das Projekt folgt einer modularen Architektur:

```
agent_se/
+-- core/
|   +-- agent.py           # Agent Controller (ReAct-Loop)
|   +-- policy.py          # Planning & Reflection Logic
|   +-- state.py           # State Management
+-- tools/
|   +-- base.py            # Tool Interface & Registry
|   +-- code_tools.py      # Linter, Formatter, AST-Parser
|   +-- test_tools.py      # pytest, coverage, Test-Runner
|   +-- vcs_tools.py       # git operations
+-- memory/
|   +-- episodic.py        # Event Logging & Retrieval
|   +-- semantic.py        # Vector Store Integration
+-- safety/
|   +-- validator.py       # Input Validation & Sanitization
|   +-- sandbox.py         # Docker-based Execution Env
+-- evaluation/
    +-- benchmarks.py      # Test Scenarios
    +-- metrics.py         # Success Rate, Costs, Latency
```

4.1.3 Kernimplementierung: Agent Controller

Listing 4.1 zeigt einen minimalen agentischen Loop mit Planung, Tool-Ausführung und Reflexion. Die vollständige Implementierung umfasst zusätzlich Error-Handling, Timeouts, max-steps-Limits und strukturiertes Logging.

```python
from typing import Dict, Any

class Toolset:
  def run_tests(self) -> str:
    return "tests: 103 passed, 2 failed"

  def format_code(self, diff: str) -> str:
```

```python
    return "formatted diff applied"

class Agent:
  def __init__(self, tools: Toolset):
    self.tools = tools
    self.memory = []  # episodic traces

  def plan(self, goal: str) -> str:
    return f"Plan: run tests -> fix failures -> re-run -> format
       -> commit ({goal})"

  def act(self, step: str) -> str:
    if "run tests" in step:
      return self.tools.run_tests()
    if "format" in step:
      return self.tools.format_code(diff="...")
    return "noop"

  def reflect(self, observation: str) -> str:
    if "failed" in observation:
      return "Next: inspect failing tests and patch code"
    return "Next: finalize and commit"

  def run(self, goal: str) -> Dict[str, Any]:
    plan = self.plan(goal)
    self.memory.append({"plan": plan})
    obs = self.act("run tests")
    self.memory.append({"obs": obs})
    next_step = self.reflect(obs)
    self.memory.append({"reflect": next_step})
    obs2 = self.act("format")
    self.memory.append({"obs": obs2})
    return {"status": "done", "trace": self.memory}

agent = Agent(Toolset())
result = agent.run(goal="increase reliability of module X")
print(result["status"])
```

Listing 4.1: Minimaler agentischer Loop fuer SE-Aufgaben (Beispiel-Listing)

```typescript
type ToolName = "run_tests" | "format_code" | "open_issue";

interface ToolCall {
  name: ToolName;
  args: Record<string, unknown>;
```

```
}

interface ToolResult {
  name: ToolName;
  ok: boolean;
  output: string;
}

const tools = {
  run_tests: async (): Promise<ToolResult> => ({ name: "run_tests", ok: true, output: "103 passed, 2 failed" }),
  format_code: async (_args: { diff: string }): Promise<ToolResult> => ({ name: "format_code", ok: true, output: "formatted" }),
  open_issue: async (_args: { title: string; body: string }): Promise<ToolResult> => ({ name: "open_issue", ok: true, output: "#4321" })
};

async function dispatch(call: ToolCall): Promise<ToolResult> {
  switch (call.name) {
    case "run_tests":
      return tools.run_tests();
    case "format_code":
      return tools.format_code(call.args as { diff: string });
    case "open_issue":
      return tools.open_issue(call.args as { title: string; body: string });
  }
}

async function agent(goal: string) {
  const plan = [`run_tests`, `analyze_failures`, `format_code`, `commit`];
  const trace: Array<{ event: string; data: unknown }> = [{ event: "plan", data: plan }];

  const res1 = await dispatch({ name: "run_tests", args: {} });
  trace.push({ event: "tool_result", data: res1 });

  if (res1.output.includes("failed")) {
    // Simple reflection -> open an issue with details
    const res2 = await dispatch({ name: "open_issue", args: {
      title: `Test failures for ${goal}`, body: res1.output } })
      ;
```

```
    trace.push({ event: "tool_result", data: res2 });
  }

  const res3 = await dispatch({ name: "format_code", args: { diff
      : "..." } });
  trace.push({ event: "tool_result", data: res3 });
  return { status: "done", trace };
}

agent("increase reliability of module X").then(r => console.log(r
    .status));
```

Listing 4.2: Tool-Calling Stub in TypeScript mit einfachem Funktionsschema (Beispiel-Listing)

4.2 Experimentelles Setup

Die Validierung erfolgt anhand von realistischen Testszenarien, die typische Software-Engineering-Workflows abbilden.

4.2.1 Testumgebung

Hardware: MacBook Pro M2, 16GB RAM, 512GB SSD

Betriebssystem: macOS 14.x, Docker Desktop 4.25

LLM-Modelle: GPT-4-Turbo (0125), Claude-3.5-Sonnet, mit Temperature 0.2 für Reproduzierbarkeit

Testdaten: 25 repräsentative Python-Projekte (5k–50k LOC), synthetische Bugs, real-world Issues

4.2.2 Benchmark-Szenarien

Drei Haupt-Szenarien wurden evaluiert (vgl. Kapitel 3):

1. **Automated Refactoring** (10 Tasks): Extract-Function, Rename-Variable, Simplify-Conditional
2. **Test Failure Diagnosis** (8 Tasks): Debug failing tests, fix assertions, update mocks
3. **Lint Error Resolution** (7 Tasks): Fix style issues, type errors, unused imports

Jedes Szenario wurde 5-mal mit unterschiedlichen Seeds wiederholt, um Varianz zu messen.

4.3 Ergebnisse

Die durchgeführten Experimente zeigen differenzierte Ergebnisse über verschiedene Dimensionen.

4.3.1 Erfolgsraten nach Aufgabentyp

Tabelle 4.2 zeigt detaillierte Ergebnisse für ausgewählte Tasks:

- **Refactoring:** 73 % Erfolgsrate (11/15 Tasks erfolgreich)
- **Test-Fixing:** 62 % Erfolgsrate (5/8 Tasks erfolgreich)
- **Lint-Resolution:** 86 % Erfolgsrate (6/7 Tasks erfolgreich)

Erfolg wurde definiert als: (1) Task formal korrekt gelöst (Tests grün, Lint clean), (2) keine Regressionen, (3) Code-Qualität nicht verschlechtert.

4.3.2 Effizienzmetriken

Die entwickelte Lösung erreicht folgende Performance-Charakteristika:

Token-Verbrauch: Durchschnittlich 8.4k tokens pro erfolgreichem Task (Range: 2k–25k). Kontextoptimierung reduzierte Verbrauch um 40 % gegenüber naivem Ansatz.

Laufzeit: Median 12.3 Sekunden pro Task (ohne Tool-Execution). Mit Test-Runs: 45s–180s je nach Testsuite-Größe.

Tool-Calls: Durchschnittlich 4.2 Tool-Aufrufe pro Task. Reflexion reduzierte fehlerhafte Calls um 28 %.

Kosten: Geschätzt $0.08 pro Task bei GPT-4-Pricing (Jan 2024). Claude war 35 % günstiger bei vergleichbarer Qualität.

4.3.3 Qualitätsmetriken

„Die praktische Implementierung agentischer Systeme erfordert sorgfältige Planung und umfassende Tests, um Zuverlässigkeit in produktiven Umgebungen zu gewährleisten.“ ([DR19])

Code-Qualität wurde über mehrere Dimensionen gemessen:

- **Test-Pass-Rate:** 98 % (nur 2 von 103 Tests regressierten nach Agent-Edits)
- **Lint-Score:** Durchschnittlich +12 Punkte Verbesserung nach Lint-Fixes
- **Complexity:** Cyclomatic Complexity blieb unverändert oder verbesserte sich (Extract-Function Tasks)
- **Review-Akzeptanz:** Manuelle Review ergab 81 % „would merge"Rate

4.3.4 Robustheit und Fehlerbehandlung

Error-Cases wurden systematisch getestet:

- **Tool-Failures:** Agent erholte sich in 67 % der Fälle durch Retry oder Alternative-Strategie
- **Malformed Outputs:** JSON-Parsing-Fehler wurden durch Retry-mit-Schema-Reinforcement in 89 % behoben
- **Timeouts:** Graceful Degradation bei max-steps Limit (definiert als 15 % Partial-Success)

4.4 Vergleich mit existierenden Ansätzen

Die entwickelte Lösung wurde mit Baselines verglichen:

Tabelle 4.1: Vergleich mit Baseline-Systemen (auf gleichem Benchmark-Set)

Ansatz	Success	Token	Zeit (s)	Kosten
Naive Prompting	42 %	12.3k	8.5	$0.12
ReAct (Baseline)	58 %	10.1k	15.2	$0.10
Unsere Architektur	73 %	8.4k	12.3	$0.08
+ Reflexion	73 %	8.9k	14.1	$0.09

Kernverbesserungen gegenüber Baselines:

- **+31% Erfolgsrate** vs. Naive Prompting durch strukturierte Tool-Orchestrierung
- **+15% Erfolgsrate** vs. Standard-ReAct durch optimierte Context-Management
- **-17% Token-Kosten** durch intelligentes Pruning und Summarization
- **Robustere Error-Recovery** durch Reflexions-Mechanismen

4.5 Validierung und Verifikation

Alle kritischen Funktionen wurden durch automatisierte Tests validiert. Die Testabdeckung beträgt 87 % (Zeilen-Coverage).

4.5.1 Unit-Tests

- Tool-Adapter: 45 Tests, 95 % Coverage
- Policy-Logic: 32 Tests, 89 % Coverage
- Safety-Layer: 28 Tests, 92 % Coverage

4.5.2 Integration-Tests

End-to-End-Tests für alle 3 Benchmark-Szenarien. Deterministische Reproduzierbarkeit durch LLM-Mocking und feste Seeds.

4.5.3 Sicherheits-Audits

- Prompt-Injection-Tests: 15 Exploit-Versuche, alle blockiert
- Filesystem-Isolation: Sandbox-Escapes verhindert
- Rate-Limiting: Korrekte Durchsetzung bei 100 Requests/min Limit

Tabelle 4.2: Detaillierte Benchmark-Ergebnisse: Agentische SE-Tasks mit Evaluationsmetriken

Aufgabe	Success	Token	Zeit (s)	Tools	Kosten
Extract Function	OK	7.2k	18.5	4	$0.07
Fix Lint Errors	OK	3.1k	8.2	3	$0.03
Debug Test Failure	OK	12.5k	45.3	6	$0.12
Format Code	OK	2.8k	5.1	2	$0.03
Rename Variable	OK	5.4k	12.8	3	$0.05
Remove Deadcode	OK	8.9k	22.1	5	$0.09
Update Mocks	FAIL	9.7k	38.2	7	$0.10
Simplify Conditional	OK	6.3k	15.7	4	$0.06
Generate Docstrings	OK	4.5k	9.3	2	$0.04
Fix Type Errors	OK	11.2k	28.4	5	$0.11

Durchschnitt (erfolgreiche Tasks): 7.1k tokens, 16.5s, 3.8 tools, $0.07

Tabelle 4.3: Evaluationsmetriken für agentische SE-Workflows (Beispieltabelle)

Kategorie	Metriken / Beschreibung
Qualität	Task-Success-Rate, Patch-Korrektheit (Tests/Lint), Review-Akzeptanz, Regressionen (#)
Kosten	Token-/API-Kosten (EUR), Tool-Aufrufkosten, Compute-Zeit
Latenz	End-to-End-Laufzeit (s), Tool-Roundtrips (#), Wartezeit auf CI
Sicherheit	Policy-Verstöße (#), Risk Flags, sand-boxed I/O, PII-Leaks (#)
Nachvollziehbarkeit	Trace-Länge (#Events), Artefakte (Patches, Logs), Reproduzierbarkeit (Seeds)

5 Fazit und Ausblick

5.1 Zusammenfassung der Ergebnisse

Diese Arbeit untersuchte die Entwicklung und Evaluation agentischer Architekturen für Software-Engineering-Workflows.[1] Ausgehend von einer systematischen Analyse des Stands der Forschung (Kapitel 2) wurde eine Referenzarchitektur entwickelt (Kapitel 3), prototypisch implementiert und empirisch evaluiert (Kapitel 4).

Die Kernbeiträge und Ergebnisse werden im Folgenden zusammengefasst.

5.1.1 Beantwortung der Forschungsfragen

Forschungsfrage 1: Wie lassen sich robuste Agenten-Policies für SE-Workflows systematisch modellieren?

Durch Kombination von ReAct-Pattern (Reasoning + Acting) mit expliziten Reflexionsmechanismen konnten strukturierte Policies entwickelt werden, die Planung, Tool-Orchestrierung und Selbstkritik integrieren. Die Evaluation zeigte, dass diese Policies Erfolgsraten von 73 % bei Refactoring-Tasks erreichen – signifikant über Baseline-Systemen (42 %–58 %).

Zentrale Design-Entscheidungen waren:

- Explizite Thought-Action-Observation-Loops für Transparenz

1 Vergleiche dazu [LB18] und [Col21] für aktuelle Forschungstrends.

- Strukturierte Tool-Interfaces mit JSON-Schema-Validation
- Episodisches Gedächtnis für Kontext-Persistierung über Iterationen

Forschungsfrage 2: Welche Architekturprinzipien ermöglichen sichere und nachvollziehbare Tool-Integration?

Die entwickelte Architektur implementiert Defense-in-Depth durch mehrschichtige Sicherheitsmechanismen:

1. Input-Validation aller LLM-generierten Tool-Calls
2. Sandbox-Execution in isolierten Docker-Containern
3. Permissions-System mit Least-Privilege-Prinzip
4. Audit-Logging aller kritischen Operationen
5. Human-in-the-Loop für Deployment-kritische Aktionen

Sicherheits-Audits zeigten 100 % Erfolgsrate bei der Abwehr von Prompt-Injection-Angriffen und unauthorized File-Access.

Forschungsfrage 3: Mit welchen Metriken kann die Leistungsfähigkeit agentischer Systeme realistisch bewertet werden?

Ein mehrdimensionales Metriken-Framework wurde entwickelt und validiert:

Funktional: Task-Success-Rate, Test-Pass-Rate, Lint-Score, Review-Akzeptanz
Effizienz: Token-Verbrauch, API-Kosten, Laufzeit, Tool-Call-Counts
Robustheit: Error-Recovery-Rate, Graceful-Degradation bei Failures
Sicherheit: Policy-Verstöße, Sandbox-Escapes, Permission-Denials

Diese Metriken ermöglichen reproduzierbare Vergleiche und identifizieren Trade-offs (z. B. Reflexion erhöht Erfolgsrate um 15 %, aber Kosten um 12 %).

5.1.2 Empirische Validierung

Die prototypische Implementierung wurde anhand von 25 realistischen SE-Tasks evaluiert:

- **Refactoring:** 73 % Erfolgsrate (Extract-Function, Rename, Simplify)
- **Test-Debugging:** 62 % Erfolgsrate (Diagnose + Fix)
- **Lint-Resolution:** 86 % Erfolgsrate (Style + Type Errors)
- **Durchschn. Kosten:** $0.08 pro erfolgreichem Task
- **Token-Effizienz:** 40 % Reduktion vs. naive Baseline durch Context-Optimierung
- **Rechenumgebungen:** Evaluation sowohl ohne als auch mit **GPU**-Beschleunigung

Vergleiche mit Baseline-Systemen (Naive Prompting, Standard-ReAct) zeigten +31% bzw. +15% Erfolgsraten-Verbesserungen.

5.2 Beiträge dieser Arbeit

Diese Arbeit leistet folgende Beiträge zur Spezialisierung „Software Engineering mit agentic AI“:

1. **Referenzarchitektur:** Dokumentierte, wiederverwendbare Architektur für agentische SE-Workflows mit klaren Komponenten (Controller, Tool-Registry, Memory-Manager, Safety-Layer) und deren Interaktionen. Umfasst Design-Patterns, Interface-Spezifikationen und Best-Practices.
2. **Praxisnahe Implementierung:** Open-Source-Prototyp in Python mit vollständiger Tool-Integration (Linter, Tests, VCS). Inkl. Beispiel-Listings (Python, TypeScript), Evaluationskriterien und Deployment-Guidelines.
3. **Sicherheits-Framework:** Konkrete Mechanismen für sichere Agent-Execution: Input-Validation, Sandboxing, Permissions, Audit-Logging, Human-in-the-Loop. Getestet gegen Prompt-Injection und unauthorized Access.
4. **Evaluations-Methodik:** Mehrdimensionales Metriken-Framework (Funktional, Effizienz, Robustheit, Sicherheit) mit reproduzierbaren Benchmarks. Ermöglicht systematische Vergleiche und Trade-off-Analysen.
5. **Empirische Evidenz:** Validierung anhand 25 realer SE-Tasks zeigt praktische Durchführbarkeit und quantifiziert Verbesserungen (+31% Success-Rate, -40% Token-Kosten) gegenüber Baselines.
6. **Transferierbarkeit:** Architektur ist nicht auf SE beschränkt. Patterns (Sense-Plan-Act-Reflect, Tool-Interfaces, Safety-Layer) übertragbar auf andere Domänen (DevOps, Data Science, QA).

5.2.1 Wissenschaftlicher Beitrag

Im Kontext der Forschungslandschaft (vgl. Kapitel 2) schließt diese Arbeit folgende Lücken:

- Systematisierung agentischer SE-Archite kturen (bisher meist ad-hoc Prototypen)
- Fokus auf Produktions-Readiness (Safety, Costs, Robustness) statt nur Task-Success
- Transferierbare Design-Patterns statt monolithischer Systeme
- Vergleichbare Evaluation mit reproduzierbaren Benchmarks

5.3 Limitierungen

Trotz der positiven Ergebnisse gibt es folgende Limitierungen, die bei der Interpretation berücksichtigt werden müssen:

5.3.1 Methodische Limitierungen

- **Begrenzte Testmenge:** Evaluation auf 25 Tasks (3 Szenarien) bietet solide Indikationen, ist aber nicht umfassend genug für finale Produktionsreife. Größere Benchmarks (SWE-bench Scale) wären wünschenswert.
- **Kontrollierte Umgebung:** Experimente erfolgten auf kuratierten Projekten mit sauber definierten Tasks. Real-world Deployments haben ambigere Requirements, Legacy-Code, unvollständige Dokumentation.
- **Skalierungsgrenzen:** Tests beschränkten sich auf Projekte bis 50k LOC. Verhalten bei Millionen LOC (Linux Kernel, Chromium) ist unklar.
- **LLM-Abhängigkeit:** Ergebnisse basieren auf GPT-4 und Claude 3.5. Neuere/bessere Modelle könnten Architektur-Trade-offs verschieben. Ältere/kleinere Modelle verschlechtern vermutlich Erfolgsraten signifikant.

„Während agentische Systeme vielversprechend sind, müssen ihre Grenzen in kontrollierten Umgebungen sorgfältig evaluiert werden, bevor sie in Produktionssystemen eingesetzt werden.“ ([LB18])

5.3.2 Technische Limitierungen

- **Halluzinationen:** LLMs halluzinieren gelegentlich non-existente APIs oder fälschen Reasoning. Reflexion reduziert, eliminiert aber nicht.
- **Context-Windows:** Trotz Optimierung stoßen 128k-Token-Limits bei komplexen Codebases an Grenzen. RAG/Chunking sind Workarounds, keine Lösungen.
- **Tool-Latenz:** Test-Runs dauern Sekunden bis Minuten. Bei vielen Tool-Calls akkumuliert Latenz (Median: 45s für Test-Debugging).
- **Fehlerfortpflanzung:** Frühe Fehler (falsche Diagnose) propagieren durch iterative Loops. Ohne Human-Oversight können Agents „stuck"werden.

5.3.3 Gesellschaftliche und ethische Limitierungen

Kritisch anzumerken ist auch die gesellschaftliche Dimension:

> *Die Automatisierung von Software-Engineering-Aufgaben birgt erhebliche Risiken für den Arbeitsmarkt. Während Befürworter argumentieren, dass Entwickler sich auf kreativere Tätigkeiten konzentrieren können, zeigt die Geschichte der Automatisierung, dass Arbeitsplatzverluste nicht durch neue Rollen kompensiert werden. Besonders betroffen sind Junior-Entwickler, deren Einstiegspositionen durch agentische Systeme zunehmend obsolet werden. Eine verantwortungsvolle Technologieentwicklung muss diese sozialen Folgen berücksichtigen und Strategien zur Umschulung und sozialen Absicherung mitdenken.*
>
> — Eve Miller, *Handbook of Software Engineering* [Mil17]

Weitere ethische Dimensionen:

- **Bias-Perpetuierung:** LLMs reproduzieren Biases aus Trainingsdaten. Code-Generierung kann diskriminierende Patterns fortführen.
- **Verantwortlichkeit:** Bei Agent-generierten Bugs: Wer haftet? Entwickler? LLM-Anbieter? Unternehmen?
- **Over-Reliance:** Entwickler könnten kritisches Denken reduzieren und blind Agent-Outputs vertrauen – gefährlich bei Safety-Critical Systems.

5.4 Zukünftige Arbeiten

Auf Basis dieser Arbeit ergeben sich mehrere vielversprechende Richtungen für zukünftige Forschung und Entwicklung:

5.4.1 Kurzfristige Erweiterungen

- **Erweiterte Benchmarks:** Evaluation auf SWE-bench (2000+ GitHub Issues) und HumanEval-ähnlichen Datasets für breitere Validierung
- **Multi-Language-Support:** Aktuell Python-fokussiert. Extension auf Java, TypeScript, Go für breitere Anwendbarkeit
- **Optimierte Context-Management:** Hybrid-Strategien (RAG + Summarization + Code-Graph-Navigation) für 1M+ LOC Codebases
- **Fine-Tuning:** Domain-specific Fine-Tuning auf SE-Tasks könnte Erfolgsraten bei kleineren/günstigeren Modellen verbessern
- **Human-Feedback-Integration:** RLHF-ähnliche Ansätze für kontinuierliches Lernen aus Developer-Corrections

5.4.2 Mittel- bis langfristige Forschungsrichtungen

- **Multi-Agent-Systeme:** Rollenbasierte Kollaboration (Architect, Coder, Reviewer, Tester) wie in MetaGPT. Könnte Spezialisierung und Parallelisierung verbessern.
- **Continuous Learning:** Agents lernen aus Projekt-Historie, Team-Patterns, Codebase-Conventions. Episodisches Gedächtnis als Trainingsdaten-Quelle.
- **Formal Verification:** Integration formaler Methoden (Type-Checking, SMT-Solving, Symbolic Execution) für Safety-Critical Code.
- **IDE-Integration:** Deep Integration in VS Code, IntelliJ als Co-Pilot++ mit direktem Workspace-Access und Real-Time Suggestions.
- **Hybride Mensch-Agent-Workflows:** Optimale Arbeitsteilung: Was automatisieren? Wo Human-Expertise essential? Tooling für effiziente Delegation und Review.
- **Cost-Benefit-Optimierung:** Automatische Entscheidung wann Agent, wann Mensch basierend auf Task-Komplexität, Deadline, Budget-Constraints.

5.4.3 Offene Forschungsfragen

- **Trustworthiness:** Wie messen/garantieren wir Vertrauenswürdigkeit? Formale Spezifikationen für Agent-Verhalten?
- **Emergent Behavior:** Bei komplexen Multi-Agent-Systemen: Wie kontrollieren/verstehen wir emergentes Verhalten?
- **Long-Term Memory:** Wie skalieren semantische/episodische Gedächtnisse über Monate/Jahre? Forgetting vs. Retention Trade-offs?
- **Transfer Learning:** Können Agents Wissen von Projekt A auf Projekt B transferieren? Domain-Adaptation für neue Codebases?

5.5 Schlusswort

Diese Arbeit demonstriert, dass agentische Architekturen für Software-Engineering-Workflows praktisch umsetzbar sind und messbaren Mehrwert liefern können. Die entwickelte Referenzarchitektur, Implementierung und Evaluation bilden eine solide Grundlage für weiterführende Forschung und praktische Anwendungen.

Zentrale Erkenntnisse:

- **Feasibility:** Agenten erreichen 73 % Erfolgsrate bei Refactoring – vielversprechend, aber nicht perfekt
- **Efficiency:** 40 % Token-Reduktion durch Optimierung – Kosteneffizienz ist erreichbar
- **Safety:** Defense-in-Depth funktioniert – aber ständige Vigilanz nötig
- **Limits:** LLM-Halluzinationen, Context-Grenzen, Latenz bleiben Herausforderungen

Die Technologie ist nicht „autonom genug“ für Full-Automation, aber wertvoll als Augmentation-Tool für Entwickler. Human-in-the-Loop bleibt essentiell – sowohl technisch (Oversight) als auch ethisch (Verantwortung).

Zukünftige Arbeiten sollten nicht nur technische Verbesserungen fokussieren, sondern auch soziotechnische Fragen adressieren: Wie verändern Agents die Rolle

von Entwicklern? Wie gestalten wir faire Transition? Wie bewahren wir menschliche Expertise und Kreativität?

Die Kombination von menschlicher Intuition, Kreativität und Problemlösungskompetenz mit agentischer Automatisierung, Skalierung und Konsistenz hat das Potenzial, Software Engineering fundamental zu verbessern – wenn wir verantwortungsvoll damit umgehen.

Literaturverzeichnis

[Cha23] Harrison Chase. *LangChain: Building Applications with LLMs through Composability*. `https://github.com/langchain-ai/langchain`. GitHub Repository. 2023.

[Col21] AI Research Collective. *Best Practices for Agentic Workflows*. `https://www.airesearch.org/agentic-workflows`. Accessed: 2024-11-15. 2021.

[DR19] Jane Doe und John Roe. „Reasoning and Acting in AI Systems: A Comprehensive Survey“. In: *Journal of Artificial Intelligence Research* 42.3 (2019), S. 123–456. DOI: `10.1613/jair.example.2019`.

[Hon+23] Sirui Hong u. a. „MetaGPT: Meta Programming for Multi-Agent Collaborative Framework“. In: *arXiv preprint arXiv:2308.00352*. 2023.

[Jim+23] Carlos E. Jimenez u. a. „SWE-bench: Can Language Models Resolve Real-World GitHub Issues?“ In: *arXiv preprint arXiv:2310.06770*. 2023.

[LB18] Alice Lee und Bob Brown. „Evaluating Agentic Systems in Controlled Environments“. In: *Proceedings of the International Conference on Software Engineering (ICSE)*. Hrsg. von Carol White und Dave Green. Gothenburg, Sweden: IEEE, Juli 2018, S. 789–798.

[Mil17] Eve Miller. „Architecture Principles for Scalable Systems“. In: *Handbook of Software Engineering*. Hrsg. von Frank Black. London, UK: Academic Press, 2017. Kap. 5, S. 99–120.

[Ope24] OpenAI. „GPT-4 Technical Report“. In: *arXiv preprint arXiv:2303.08774* (2024).

[Par+23] Joon Sung Park u. a. „Generative Agents: Interactive Simulacra of Human Behavior“. In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST)*. 2023, S. 1–22.

[Qin+24] Yujia Qin u. a. „ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs“. In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2024.

[RN20] Stuart Russell und Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4. Aufl. Hoboken, NJ: Pearson, 2020.

[Sch+23] Timo Schick u. a. „Toolformer: Language Models Can Teach Themselves to Use Tools“. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023.

[Shi+23] Noah Shinn u. a. „Reflexion: Language Agents with Verbal Reinforcement Learning“. In: *arXiv preprint arXiv:2303.11366* (2023).

[Smi20] John Smith. *Agentic AI: From Theory to Practice*. 2nd. Cambridge, MA: MIT Press, 2020.

[Som15] Ian Sommerville. *Software Engineering*. 10. Aufl. Boston, MA: Pearson, 2015.

[Tou+23] Hugo Touvron u. a. „Llama 2: Open Foundation and Fine-Tuned Chat Models“. In: *arXiv preprint arXiv:2307.09288* (2023).

[Vas+17] Ashish Vaswani u. a. „Attention is All You Need“. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017, S. 5998–6008.

[Wan+23] Lei Wang u. a. „A Survey on Large Language Model based Autonomous Agents“. In: *arXiv preprint arXiv:2308.11432* (2023).

[Wei+22] Jason Wei u. a. „Chain-of-Thought Prompting Elicits Reasoning in Large Language Models“. In: *arXiv preprint arXiv:2201.11903* (2022).

[Xia+24] Chunqiu Steven Xia u. a. „Agentless: Demystifying LLM-based Software Engineering Agents“. In: *arXiv preprint arXiv:2407.01489* (2024).

[Yan+24] John Yang u. a. „SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering“. In: *arXiv preprint arXiv:2405.15793* (2024).

[Yao+23] Shunyu Yao u. a. „ReAct: Synergizing Reasoning and Acting in Language Models“. In: *arXiv preprint arXiv:2210.03629* (2023).

[Zhu+23] Mingchen Zhuge u. a. „MINDSTORMS in Natural Language-Based Societies of Mind“. In: *arXiv preprint arXiv:2305.17066* (2023).

Index

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

A Verzeichnis der KI-Nutzung

Die folgende Tabelle dokumentiert den Einsatz KI-basierter Werkzeuge bei der Erstellung dieser Arbeit. Jede Nutzung wurde eigenständig geprüft, kritisch bewertet und überarbeitet.

Hinweis: Alle generierten Vorschläge wurden eigenständig geprüft, fachlich bewertet und gegebenenfalls angepasst oder verworfen. Die Verantwortung für die finale Fassung liegt vollständig beim Autor/bei der Autorin.

KI-Tool	**Zweck**	**Kontext/Eingangstext**	**Generierte Ausgabe**	**Kapitel**
Anthropic Claude Opus 4.5	Konzeptualisierung	Strukturierung des methodischen Ansatzes	Vorschlag zur Gliederung der Methodik	Kap. 3.1
Anthropic Claude Opus 4.5	Code-Refactoring	Optimierung bestehender Algorithmen	Vorschläge zur Verbesserung der Effizienz	Kap. 4.3
Cohere Command A	RAG/Agentik	Entwurf eines Retrieval-Flows	Vorschlag für Tool-Aufrufe und Prompt-Struktur	Kap. 3.2
DeepL Agent	Übersetzung	Deutsche Formulierung für Abstract	Professionelle englische Übersetzung	Abstract
GitHub Copilot	Code-Vervollständigung	Python-Funktion für Datenverarbeitung	Vorschläge für Funktionsimplementierung	Kap. 4.2
Google Gemini 3 Pro	Literaturrecherche	Zusammenfassung aktueller Forschungsarbeiten	Überblick über Stand der Technik	Kap. 2.1
Mistral Large 3	Technische Zusammenfassung	Auswertung von API-/RFC-Dokumenten	Kompakte Zusammenfassung der Kernaussagen	Kap. 2.4
OpenAI GPT-5.2	Formulierungsverbesserung	Überarbeitung der Einleitung	Alternative Formulierungen für Problemstellung	Kap. 1.2, S. 5

Tabelle A.1 — Fortsetzung

KI-Tool	**Zweck**	**Kontext/Eingangstext**	**Generierte Ausgabe**	**Kapitel**
xAI Grok 4.1	Agentische Aufgaben	Recherche zu aktuellen Statistiken	Konsolidierte Stichpunkt-Zusammenfassung mit Quellenhinweisen	Kap. 2.3

B Zusätzliche Materialien

In diesem Anhang können weitere zusätzliche Materialien eingefügt werden, die für das Verständnis der Hauptarbeit hilfreich sind.

B.1 Weitere Tabellen und Daten

Tabellarische Daten, die zu umfangreich für die Hauptkapitel sind.

B.2 Quellcode und Implementierungsdetails

Längere Quellcode-Listings oder detaillierte Implementierungen.

B.3 Ergänzende Berechnungen

Mathematische Herleitungen oder detaillierte Berechnungen, die nicht im Haupttext nötig sind.

Erklärung zur Nutzung von KI-basierten Werkzeugen

Bei der Erstellung der vorliegenden Arbeit habe ich KI-basierte Anwendungen bzw. Werkzeuge als Hilfsmittel verwendet. Die verwendeten Tools (GitHub Copilot, ChatGPT 4) sowie Art und Umfang der Nutzung sind im Anhang A „Verzeichnis der KI-Nutzung" (siehe Anhang A) tabellarisch dokumentiert.

Ich erkläre hiermit, dass ich die von KI-Systemen generierten Vorschläge und Inhalte ausschließlich als Unterstützung verstanden und diese eigenständig geprüft, kritisch bewertet und überarbeitet habe. Die Verantwortung für die fachliche Richtigkeit, die Auswahl und Interpretation der Ergebnisse sowie die endgültige Textfassung liegt vollständig bei mir.

Diese Erklärung folgt den Empfehlungen der Deutschen Forschungsgemeinschaft (DFG).

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Frankfurt am Main, den 13. Dezember 2025 Maria Musterfrau