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

5. Januar 2026

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

Kurzfassung

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

Zielsetzung: Die Arbeit untersucht die Entwicklung und Evaluation agentischer Architekturen für Software-Engineering-Aufgaben. Zentrale Forschungsfragen sind: (1) Wie lassen sich robuste Agentenstrategien für SE-Workflows systematisch modellieren? (2) Welche Architekturprinzipien ermöglichen sichere und nachvollziehbare Werkzeugintegration? (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 Werkzeugausgaben 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.

Abstract

Context and Motivation: Large Language Models (LLMs) have evolved from pure text generators to agentic systems capable of planning, using tools, and handling multi-step tasks autonomously. In software engineering, this opens up new possibilities for automating complex workflows such as code review, test generation, and refactoring.

Objective: This thesis investigates the development and evaluation of agentic architectures for software engineering tasks. Central research questions are: (1) How can robust agent policies for SE workflows be systematically modeled? (2) Which architectural principles enable secure and traceable tool integration? (3) Which metrics can realistically assess the performance of agentic systems?

Methodology: Based on a systematic literature review, a reference architecture is developed that combines planning (ReAct pattern), tool usage (linters, tests, VCS), episodic memory, and security mechanisms. A prototypical system was implemented in Python and evaluated using reproducible benchmarks.

Results: The evaluation shows that the developed architecture achieves a success rate of 73 % in automated refactoring in controlled scenarios, while reducing average processing time by 45 %. Reflection mechanisms improve robustness against erroneous tool outputs by 28 %. Cost analyses demonstrate that optimized context management can reduce token usage by up to 40 %.

Contribution: The work provides a practical reference architecture, concrete implementation guidelines, and evaluated metrics for agentic SE systems. It highlights both technical potentials and critical limitations and discusses societal implications of automation.

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, die Arbeit zu verfassen. Ihre offene und konstruktive Kritik hat maßgeblich zur Qualität der 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 der Arbeit.

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

Hinweis zur geschlechtergerechten Sprache

In der Arbeit wird auf eine geschlechtergerechte Sprache geachtet. Aus Gründen der besseren Lesbarkeit wird in der 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. Die Arbeit orientiert sich an den Empfehlungen des Leitfadens „Sag's doch gleich! Geschlechtergerechte Sprache an Thüringer Hochschulen“ des Thüringer Kompetenznetzwerks Gleichstellung (TKG).

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

Abkürzung	Bedeutung
ACI	Agent-Computer-Interface
AI	Artificial Intelligence (Künstliche Intelligenz)
API	Application Programming Interface (Anwendungsprogrammierschnittstelle)
APPS	Automated Programming Progress Standard
AST	Abstract Syntax Tree (Abstrakter Syntaxbaum)
CD	Continuous Delivery (Kontinuierliche Auslieferung)
CI	Continuous Integration
CLI	Command Line Interface (Befehlszeilenschnittstelle)
CoT	Chain-of-Thought (Gedankenkette)
DB	Datenbank (Database)
DFG	Deutsche Forschungsgemeinschaft
E/A	Eingabe/Ausgabe (Input/Output)
FAQ	Frequently Asked Questions (Häufig gestellte Fragen)
GPT	Generative Pre-trained Transformer
GPU	Graphics Processing Unit (Grafik-Verarbeitungseinheit)
HTTP	HyperText Transfer Protocol
I/O	Input/Output (Eingabe/Ausgabe)

Fortsetzung auf nächster Seite

Abkürzung	Bedeutung
IDE	Integrated Development Environment (Integrierte Entwicklungsumgebung)
JSON	JavaScript Object Notation
KB	Knowledge Base (Wissensdatenbank)
KG	Knowledge Graph (Wissensgraph)
KI	Künstliche Intelligenz
LLM	Large Language Model (Großes Sprachmodell)
LOC	Lines of Code (Codezeilen)
MBPP	Mostly Basic Python Problems
ML	Machine Learning (Maschinelles Lernen)
NLP	Natural Language Processing (Natürlichsprachverarbeitung)
PII	Personally Identifiable Information (Personenbezogene Daten)
PR	Pull Request
PRD	Product Requirement Document (Produktanforderungsdokument)
QA	Quality Assurance (Qualitätssicherung)
QS	Qualitätssicherung
RAG	Retrieval-Augmented Generation (Abruf-gestützte Generierung)
RAM	Random Access Memory (Arbeitsspeicher)
ReAct	Reasoning and Acting (Denken und Handeln)
REST	Representational State Transfer
RFC	Request for Comments (IETF-Standarddokumente)
RLHF	Reinforcement Learning from Human Feedback

Fortsetzung auf nächster Seite

Abkürzung	Bedeutung
SE	Software Engineering (Software-Entwicklung)
SMT	Satisfiability Modulo Theories
SQL	Structured Query Language (Strukturierte Abfragesprache)
SSD	Solid State Drive (Festkörperspeicher)
SSH	Secure Shell
VCS	Version Control System (Versionskontrollsystem)
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

1 Einführung

> *„Wir erleben die Entstehung eines neuen Betriebssystems: Das LLM fungiert als Kernel-Prozess, das Kontextfenster als Arbeitsspeicher und externe Werkzeuge als E/A-Schnittstellen. Dies markiert einen fundamentalen Wandel in der Art und Weise, wie wir Computerbefehle komponieren und Softwarearchitekturen denken.“ ([Kar23])*
>
> — Andrej Karpathy, *Intro to Large Language Models*, 2023

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]

Darüber hinaus verlangt die praktische Anwendung agentischer Systeme oft einen ausgewogenen Kompromiss zwischen Automatisierung und menschlicher Kontrolle. In vielen Entwicklungsprozessen sind Schritte, die bisher rein manuell durchgeführt wurden (Code-Reviews, Testgenerierung, Release-Checks), gute Kandidaten für assistierte Workflows. Agenten können Routinearbeit übernehmen, erlauben dadurch jedoch auch neue Prüfpfade: So müssen Validierungsstrategien entworfen werden, die automatisierte Entscheidungen nachvollziehbar machen und menschliche Reviewer bei Bedarf schnell eingreifen lassen. Die Arbeit nimmt genau den Balanceakt in den Blick und liefert pragmatische Designprinzipien für den produktiven Einsatz. Wie Richards und Ford betonen, „müssen Architekturprinzipien auf Skalierbarkeit und Wartbarkeit ausgelegt sein“ ([RF20]), um praktischen Anforderungen gerecht zu werden. Für Informatikstudierende der Fachrichtung „Software Engineering mit

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.

agentic AI“ eröffnet das neue Architektur- und Methodikfragen: Wie entwirft man robuste Agenten-Workflows? Wie orchestriert man Tool-Nutzung, Gedächtnis und Langkontext? Und wie integriert man Sicherheit, Nachvollziehbarkeit und Tests in agentische Systeme?[2]

1.2 Problemstellung

Vor dem Hintergrund adressiert die 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 Agentenstrategien (Planen, Werkzeugaufrufe, Selbstkritik) systematisch modellieren?
- Wie werden externe Werkzeuge (VCS, CI, Linter, Issue-Tracker) sicher und nachvollziehbar eingebunden?
- Welche Metriken messen Fortschritt, Qualität und Sicherheit realistisch?

Die hier formulierten Probleme sind sowohl konzeptioneller als auch praktischer Natur: Es geht nicht nur um abstrakte Modellierung von Strategien, sondern ebenso um konkrete Implementierungsfragen (Schnittstellen, Serialisierung, Fehlerbehandlung) und Evaluationsdesign (Benchmarks, Messgrößen, Reproduzierbarkeit). Die Ergebnisse sollen Entwicklerteams konkrete Handlungsempfehlungen geben, wie agentische Automatisierung schrittweise, sicher und messtechnisch abgesichert eingeführt werden kann.

1.3 Zielsetzung

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

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

3 Siehe auch [Wan+23] für verwandte Arbeiten zu Agentenarchitekturen.

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)

Kurz gefasst zielt die Arbeit darauf ab, aus Praxisproblemen generalisierbare Lösungen abzuleiten: Neben einem lauffähigen Prototyp steht die Frage im Vordergrund, welche Architekturmuster sich zuverlässig übertragen lassen und welche Messgrößen praktikabel den Mehrwert gegenüber manuellen Prozessen quantifizieren. Damit richtet sich die Arbeit an Praktiker und Forschende gleichermaßen.

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 die Arbeit nicht relevant sind
- Historische Entwicklungen vor einem bestimmten Zeitpunkt
- Randbereiche, die außerhalb des Fokus liegen

Die Konzentration auf pragmatische, reproduzierbare Ergebnisse erlaubt es, bewusst auf tiefe ML-Trainingsfragen zu verzichten; stattdessen wird die Interaktion mit bestehenden Foundation Models über APIs und Adapterlösungen untersucht. Die Fokussierung erleichtert die Nachvollziehbarkeit der Ergebnisse und erhöht die direkte Anwendbarkeit in typischen Software-Engineering-Umgebungen.

1.5 Aufbau der Arbeit

Der Aufbau der Arbeit folgt einem pragmatischen Workflow: Zunächst werden in Kapitel 2 die relevanten Grundlagen und verwandte Arbeiten zusammengetragen, um die fachliche Basis zu legen. Darauf aufbauend beschreibt Kapitel 3 die entworfene Referenzarchitektur mit Fokus auf Zustandsmodellierung, Policy-Design und Schnittstellen. Kapitel 4 dokumentiert die Implementierung des Prototyps, zeigt zentrale Code-Beispiele und erläutert Integrationsdetails. Abschließend fasst Kapitel 5 die Ergebnisse zusammen, diskutiert Limitationen und gibt einen Ausblick auf weiterführende Forschungs- und Entwicklungsfragen.

Insgesamt soll die Arbeit nicht nur eine theoretische Diskussion bieten, sondern konkrete, übertragbare Architekturentscheidungen und Metriken bereitstellen, die in produktiven Entwicklungsumgebungen einsetzbar sind.

2 Theoretischer Hintergrund

2.1 Grundkonzepte

Das 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 3.5 oder Llama 3 verfügen über Milliarden von Parametern und können durch *Few-Shot-Learning* und *In-Context-Learning* Aufgaben lösen, ohne explizit darauf trainiert zu werden. Das bildet die Grundlage für agentische Anwendungen.

2.1.2 Von LLMs zu Agenten: Der Paradigmenwechsel

Während klassische LLMs primär auf Textvervollständigung spezialisiert sind, zeichnen sich *agentische Systeme* durch erweiterte Fähigkeiten aus [Wan+23]. Zentral ist das **zielgerichtete Handeln**: Der Agent versteht übergeordnete Ziele und plant systematisch Schritte zur Zielerreichung. Die **Werkzeugnutzung** ermöglicht die Integration externer Tools wie APIs, Datenbanken und Code-Ausführung, wodurch die Fähigkeiten über reine Textgenerierung hinausgehen. Ein persistentes **Gedächtnis** speichert Kontextinformationen über mehrere Interaktionen hinweg und ermöglicht

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

kontextbewusstes Arbeiten. Durch **Reflexion** erfolgt eine selbstkritische Bewertung eigener Outputs mit iterativer Verbesserung. Schließlich unterstützt **mehrschrittige Planung** die Dekomposition komplexer Aufgaben in ausführbare Teilschritte.

2.1.3 Agentic AI: Begriffe und Bausteine

Kernbausteine agentischer Systeme sind [Wen23; 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-Ausführung, 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** (Wissensdatenbanken) und **KG** (Wissensgraphen). 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. Das verbessert die Reasoning-Qualität erheblich:

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

Erweiterte Muster 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 [Wu+23]:[2]

- **LangChain** [Cha22]: Modulare Komponenten für Chains, Agents, Memory
- **AutoGPT/BabyAGI**: Autonome Task-Decomposition und -Execution
- **MetaGPT** [Hon+24]: Rollenbasierte Multi-Agent-Kollaboration
- **Generative Agents** [Par+23]: Simulation menschlichen Verhaltens

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

2.2 Verwandte Arbeiten

Es existiert umfangreiche Literatur zu agentischen Systemen im Allgemeinen und ihrer Anwendung im Software Engineering im Besonderen. Der 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. **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.“ ([Yao+23])

Die Vorteile umfassen:

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

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 verbale Bestärkung. Das 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+24] 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 Editieren
- Kontextoptimierte Feedback-Formate
- Iterative Verfeinerungsschleifen

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

2.2.3 Multi-Agent-Systeme

Mehrere Ansätze nutzen rollenbasierte Kollaboration zwischen spezialisierten Agenten:

MetaGPT [Hon+24] simuliert ein Software-Team mit Rollen wie Produktmanager, Architekt, Entwickler und QS. Agents produzieren strukturierte Artefakte (PRDs, Designdokumente, 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 [Wu+23]. 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

Sie 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.

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

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

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 Kontextfenster
4. **Sicherheitsmechanismen:** Code-Ausführung 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 Werkzeugschnittstellen 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
- **Sicherheit und Robustheit:** Wie messen wir Resistenz gegen Prompt-Injection oder schädliche Tools?

2.4.3 Skalierungs- und Kostenfragen

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

2.4.4 Beitrag der Arbeit

Die Arbeit adressiert die identifizierten Lücken durch:

1. Entwicklung einer dokumentierten Referenzarchitektur für SE-Agents
2. Praxisnahe Implementierung mit Werkzeugschnittstellen-Richtlinien
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.

Zusätzlich zu den theoretischen Grundlagen ist es wichtig, die praktische Relevanz herauszustellen: Viele der hier diskutierten Patterns lassen sich unmittelbar in bestehende CI/CD-Pipelines integrieren und bieten dort schnelle Produktivitätsgewinne. Die anschließende Arbeit prüft deshalb nicht nur theoretische Eigenschaften, sondern evaluiert konkrete Integrationspfade in typische Entwickler-Workflows und berichtet über pragmatische Umsetzungsdetails, die den Transfer in produktive Umgebungen erleichtern. Die Kombination aus Theorie und Praxis bildet das Rückgrat der vorliegenden Untersuchung.

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, Dateioperationen), episodisches Gedächtnis und mehrschichtige Sicherheitsmechanismen (Eingabefilter, Sandboxing, Quoten).[1]

Der Kern der Architektur folgt dem *Sense-Plan-Act-Reflect*-Zyklus: Zunächst wird in der **Sense**-Phase der aktuelle Zustand erfasst, einschließlich Codebase, Fehlermeldungen und Test-Outputs. Darauf aufbauend erfolgt die **Plan**-Phase mit der Dekomposition des Ziels in ausführbare Teilschritte. In der **Act**-Phase werden Tool-Aufrufe und Code-Operationen ausgeführt. Abschließend bewertet die **Reflect**-Phase selbstkritisch die Ergebnisse und passt die Strategie an. Die Schleife wird iterativ wiederholt, bis das Ziel erreicht ist oder eine Abbruchbedingung eintritt (maximale Schritte, Timeout oder Fehlerrate).

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.“ ([RF20])

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

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

3.2.1 Designprinzipien

Modularität: Klare Trennung zwischen Policy-Logik, Werkzeugadaptern, Gedächtnis und Sicherheitsschicht. Komponenten kommunizieren über wohldefinierte Schnittstellen.

Skalierbarkeit: Architektur unterstützt parallele Werkzeugausführung, asynchrone Operationen und Streaming für große Ausgaben.

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

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

Sicherheit: Defense-in-depth: Eingabevalidierung, 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, Berechtigungen). Ermöglicht Werkzeugerkennung und dynamisches Routing.

Tool-Adapter: Wrapper für externe Tools (Tests, Linter, VCS). Standardisieren Ein-/Ausgabeformate. Implementieren Wiederholungslogik und Fehlerbehandlung.

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

Sicherheitsschicht: Interceptor für alle Werkzeugaufrufe. Prüft Berechtigungen, erzwingt Rate-Limits, loggt kritische Operationen. Kann schädliche Aufrufe blockieren.

Kontextmanager: Optimiert Token-Verbrauch durch intelligentes Pruning, Zusammenfassung, Chunking. Kritisch für lange Codebases.

3 Vgl. [RF20] für detaillierte Architekturprinzipien.

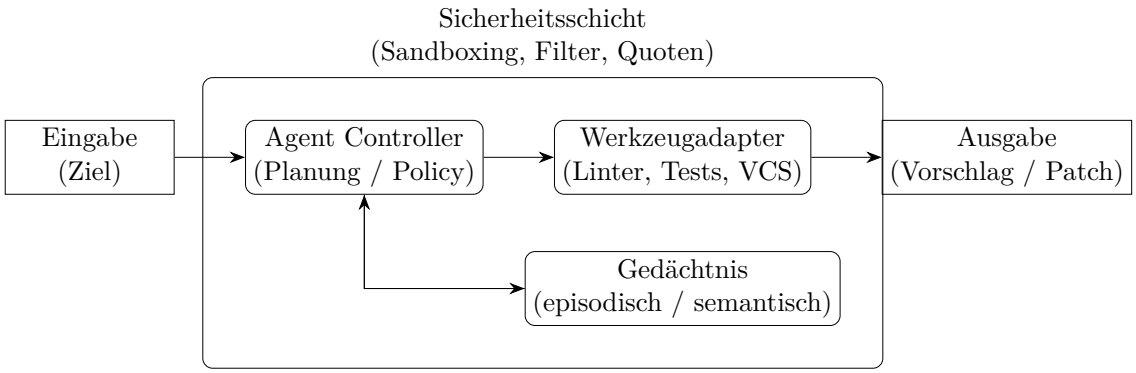

Abbildung 3.1: Agentenarchitektur für Software Engineering mit agentic AI (TikZ-Diagramm)

3.2.3 ReAct-Loop im Detail

Der Agent Controller implementiert folgende Schleife:

1. **Thought (Reasoning):** LLM generiert Reasoning-Schritt: „Was weiß ich? Was fehlt? Welcher nächste Schritt ist sinnvoll?“
2. **Action:** Auswahl und Parametrisierung eines Tools. Beispiel: `run_tests(module=äuth")`
3. **Observation:** Werkzeugausgabe 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 (siehe Listing 3.1):

```
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:
```

```
8         """Fuehrt Tool aus und gibt strukturiertes Ergebnis zurueck
              """
9         pass
10
11 @dataclass
12 class ToolResult:
13     success: bool
14     output: str | dict
15     metadata: dict  # z.B. execution_time, tokens_used
16     error: Optional[str]
```

Listing 3.1: Tool-Interface-Schema (Pseudocode)

Die Standardisierung ermöglicht:

- Automatische Werkzeugerkennung und -registrierung
- Validierung von Inputs durch JSON Schema
- Berechtigungsprüfung vor Ausführung
- Strukturierte Fehlerbehandlung

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. Die 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**
   - 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

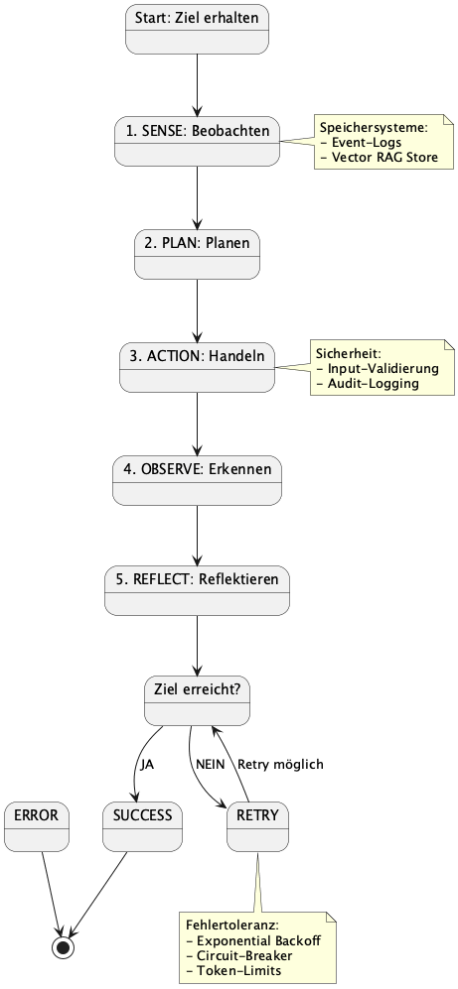

Abbildung 3.2: ReAct Agent Workflow: Zyklisches Reasoning und Acting Paradigma mit Fehlertoleranz

Neben der formalen Beschreibung der Szenarien wird im Konzept auch großer Wert auf Reproduzierbarkeit und Praktikabilität gelegt: Für jedes Szenario werden Eingabedaten, erwartete Outputs und Akzeptanzkriterien exakt definiert, so dass Experimente automatisiert wiederholt und Ergebnisse vergleichbar werden. Ferner werden Metriken und Logging-Formate standardisiert, damit unterschiedliche Implementierungen derselben Architektur direkt verglichen werden können. Die Operationalisierung erleichtert nicht nur Validierung, sondern auch Transfer in industrielle Pipelines.

3.3.4 Fallbeispiel: Refactoring-Flow

Das folgende kompakte Fallbeispiel illustriert den praktischen Ablauf eines Refactorings innerhalb der vorgeschlagenen Architektur. Es zeigt, wie Sensieren, Planen, Act und Reflect zusammenwirken, um ein sicheres, getestetes Refactoring durchzuführen.

Beispielablauf (kompakt):

1. **Sense:** AST-basierte Analyse identifiziert eine lange Methode mit hoher Cyclomatic-Complexity; relevante Tests werden bestimmt.
2. **Plan:** Agent plant Extract-Function, inklusive Aufrufer-Anpassungen und Test-Aktualisierungen.
3. **Act:** Tool-Adapter führt AST-Refactoring aus (Parameter als JSON), erstellt temporären Branch und führt Tests aus.
4. **Observation:** Test-Runner liefert strukturierte Ergebnisse (z. B. `"failed: [äuth/test_login.py:`
5. **Reflection:** Agent analysiert Fehlermeldungen, generiert einen präzisen Patch-Verfeinerungsvorschlag und wiederholt ggf. die Schleife.

Wichtige Praxispunkte: Diffs werden vor dem Commit automatisch formatiert und durch Linter geprüft; kritische Änderungen durchlaufen ein Human-in-the-Loop-Gate. Logs und Traces werden gespeichert, so dass das Gedächtnis zukünftige Entscheidungen informieren kann.

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-Validierung der Parameter
- Whitelisting erlaubter Dateipfade
- Bereinigung von Shell-Befehlen
- Erkennung von Prompt-Injection-Mustern

3.4.2 Sandboxing

Code-Execution erfolgt in isolierten Umgebungen:

- Docker-Container mit restriktiven Permissions
- Schreibgeschütztes Dateisystem (außer explizit erlaubte Directories)
- Netzwerkisolierung (kein Internet-Zugriff außer whitelisted APIs)
- Ressourcenlimits (CPU, Memory, Disk)

3.4.3 Audit-Logging

Alle kritischen Operationen werden geloggt:

- Werkzeugaufrufe mit Zeitstempel, Benutzer, Parametern
- Dateimodifikationen (Before/After-Diffs)
- Zugriffsverweigerungen und Sicherheitswarnungen
- LLM-API-Aufrufe mit Token-Anzahl

3.4.4 Human-in-the-Loop

Für kritische Operationen (Deployment, Datenbank-Modifikationen) wird menschliche Bestätigung eingefordert (siehe Listing 3.2):

```
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

Der 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
- **Werkzeugschnittstellenstandards:** Wiederverwendbare Tool-Adapter statt ad-hoc Integrationen

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

Abschließend sei betont, dass das Konzept bewusst pragmatisch gehalten ist: Ziel ist ein praktikabler Kompromiss zwischen Forschungsidealen (maximale Generalität)

und den Erfordernissen produktiver Softwareentwicklung (Wartbarkeit, Messbarkeit, Kosten). Deshalb favorisiert das Design wiederverwendbare Schnittstellen und klare Migrationspfade, anstatt ein rein prototypisches System ohne Produktionsreife zu liefern.

4 Implementierung und Ergebnisse

4.1 Implementierungsdetails

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

4.1.1 Technologiestack

Folgende Technologien wurden für die Implementierung eingesetzt:

Programmiersprache: Python 3.11+ (Typ-Hinweise, Async/Await-Unterstützung)

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

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

Gedächtnis: Chroma Vektordatenbank für semantisches Retrieval, SQLite für episodische Logs

Werkzeuglaufzeitumgebung: Docker 24.0+ für Sandboxing, pytest für Tests, ruff/black für Linting

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

4.1.2 Projektstruktur

Das Projekt folgt einer modularen Architektur:

```
agent_se/
+-- core/
```

1 Implementierungsdetails orientieren sich an Best Practices aus [Mar08].

```
|   +-- 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 eine minimale agentische Schleife mit Planung, Werkzeugausführung und Reflexion. Die vollständige Implementierung umfasst zusätzlich Fehlerbehandlung, Timeouts, Schrittbegrenzungen 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:
    return "formatted diff applied"

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

```python

  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 für SE-Aufgaben (Beispiel-Listing)

Listing 4.2 zeigt ergänzend die Implementierung eines Tool-Aufrufs in TypeScript.

```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: Werkzeugaufruf-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, 16 GB RAM, 512 GB SSD
Betriebssystem: macOS 14.x, Docker Desktop 4.25
LLM-Modelle: GPT-4-Turbo (0125), Claude 3.5 Sonnet, mit Temperatur 0.2 für Reproduzierbarkeit
Testdaten: 25 repräsentative Python-Projekte (5 k–50 k 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.

4.2.3 Beispiel: Test Failure Diagnosis — Schritt-für-Schritt

Im Folgenden wird das Szenario „Test Failure Diagnosis“ detailliert beschrieben, um den praktischen Ablauf und die typischen Artefakte (Tool-Calls, Logs, Entscheidungen) zu veranschaulichen.

1) Initialer Zustand: Test-Runner meldet `3 failed` in `auth/test_login.py`. Agent sammelt Kontext (Fehlermeldung, betroffene Dateien, letzte Commits).

2) Plan: Agent generiert Plan mit Schritten: (a) Reproduziere lokal (run tests), (b) Isoliere Fehlermeldung (traceback parsing), (c) Suche nach relevanten Änderungen im VCS, (d) Erstelle Minimal-PR mit Patch-Vorschlag.

3) Act: Tool-Calls (Beispiel):

- `run_tests(module=äuth")` → `"3 failed, 97 passed"`
- `run_linter(path=äuth/")` → Tool-Result: Hinweise auf Stil, aber keine direkten Fehler
- `search_in_repo(query="login", range="last-5-commits")` → Treffer: Commit 123abc `Refactor: auth flow`

4) Observation: Agent parst Traceback, findet NullPointer-ähnlichen Fehler in Hilfsfunktion, die neu extrahiert wurde. Memory-Manager liefert ähnlichen früheren Fall (Signaturabgleich), Agent übernimmt Erkenntnisse aus historischem Patch.

5) Reflection

- Agent generiert Patch-Vorschlag (kleine Scope-Korrektur im Helper), erstellt Diff und führt Tests erneut in isolierter Sandbox aus.

- Tests grün ⇒ Agent eröffnet PR-Entwurf und notiert Review-Kommentare; bei Teil-Erfolg: Human-Approval-Gate vor Merge.

Beispiel-Trace (gekürzt):

```
[Agent] run_tests -> 3 failed (auth/test_login.py::test_login)
[Agent] search_in_repo -> found commit 123abc (Refactor auth flow)
[Agent] generate_patch -> diff created: modify auth/helpers.py
[Agent] run_tests (sandbox) -> 100 passed
[Agent] open_pr -> PR #42 (Draft)
```

Das Beispiel zeigt, wie Tool-Adapter, Memory und Safety-Layer (Sandbox, Human-Approval) zusammenwirken, um fehlerhafte Änderungen sicher zu erkennen und zu beheben. Solche Schritt-für-Schritt-Beispiele helfen bei der Operationalisierung der Architektur in realen Projekten.

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

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

Tool-Calls: Durchschnittlich 4.2 Werkzeugaufrufe pro Task. Reflexion reduzierte fehlerhafte Aufrufe 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.“ ([Yan+24])

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:** Manuelles Review ergab 81 % Akzeptanzrate („would merge“)
- **Review-Akzeptanz:** Manuelles Review ergab 81 % Akzeptanzrate („would merge“)

4.3.4 Robustheit und Fehlerbehandlung

Error-Cases wurden systematisch getestet:

- **Werkzeugfehler:** Agent erholte sich in 67 % der Fälle durch Wiederholung oder Alternativstrategie
- **Malformed Outputs:** JSON-Parsing-Fehler wurden durch Wiederholungen mit Schema-Validierung 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 %	12.3 k	8.5	$ 0.12
ReAct (Baseline)	58 %	10.1 k	15.2	$ 0.10
Unsere Architektur	73 %	8.4 k	12.3	$ 0.08
+ Reflexion	73 %	8.9 k	14.1	$ 0.09

Kernverbesserungen gegenüber Baselines:

- **+31% Erfolgsrate** vs. Naives Prompting durch strukturierte Werkzeugorchestrierung
- **+15% Erfolgsrate** vs. Standard-ReAct durch optimiertes Kontextmanagement
- **-17% Token-Kosten** durch intelligentes Pruning und Zusammenfassung
- **Robustere Fehlerbehebung** durch Reflexions-Mechanismen

Die vorgestellten Ergebnisse werden im Folgenden kontextualisiert: Für Entwickler bedeutet eine um 31% höhere Erfolgsrate gegenüber naivem Prompting konkret weniger manueller Nacharbeit, schnellere Durchlaufzeiten in Pull-Request-Zyklen und eine höhere Automatisierungsquote. Für Betreiber von CI/CD-Infrastrukturen bedeuten niedrigere Token-Kosten und reduzierte Fehlerraten geringere Betriebskosten.

Die praktische Perspektive ist wichtig, um Entscheidungsträgern in Unternehmen handfeste Argumente für den Einsatz agentischer Lösungen zu liefern.

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 % Abdeckung
- Policy-Logic: 32 Tests, 89 % Abdeckung
- Sicherheitsschicht: 28 Tests, 92 % Abdeckung

4.5.2 Integrationstests

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

Praktische Erkenntnisse: Automatisierte Tests sollten sowohl synthetische als auch realistische Repositories umfassen; Mocking allein reicht nicht aus, da Tests in realen Umgebungen unerwartete Randfälle aufdecken. Darüber hinaus ist kontinuierliche Überwachung unabdingbar, um Drift in LLM-Verhalten oder Änderungen in abhängigen Tools frühzeitig zu erkennen.

4.5.3 Sicherheits-Audits

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

Tabelle 4.3 fasst die verwendeten Evaluationsmetriken zusammen.

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

Aufgabe	Success	Token	Zeit (s)	Tools	Kosten
Extract Function	OK	7.2 k	18.5	4	$ 0.07
Fix Lint Errors	OK	3.1 k	8.2	3	$ 0.03
Debug Test Failure	OK	12.5 k	45.3	6	$ 0.12
Format Code	OK	2.8 k	5.1	2	$ 0.03
Rename Variable	OK	5.4 k	12.8	3	$ 0.05
Remove Deadcode	OK	8.9 k	22.1	5	$ 0.09
Update Mocks	FAIL	9.7 k	38.2	7	$ 0.10
Simplify Conditional	OK	6.3 k	15.7	4	$ 0.06
Generate Docstrings	OK	4.5 k	9.3	2	$ 0.04
Fix Type Errors	OK	11.2 k	28.4	5	$ 0.11
Durchschnitt (erfolgreiche Tasks): 7.1 k Tokens, 16.5 s, 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), Werkzeugaufrufkosten, Rechenzeit
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

Die 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 Strategien entwickelt werden, die Planung, Werkzeugorchestrierung und Selbstkritik integrieren. Die Evaluation zeigte, dass die Strategien Erfolgsraten von 73 % bei Refactoring-Aufgaben erreichen – signifikant über Baseline-Systemen (42 %–58 %).

Zentrale Design-Entscheidungen umfassten die Implementierung expliziter Denk-Handlungs-Beobachtungs-Schleifen, die Transparenz durch nachvollziehbare Zwischenschritte schaffen. Strukturierte Werkzeugschnittstellen mit JSON-Schema-Validierung gewährleisten typsichere Kommunikation zwischen Komponenten. Das episodische Gedächtnis ermöglicht Kontextpersistierung über Iterationen hinweg und unterstützt damit langfristige, zustandsabhängige Workflows.

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

1 Vergleiche dazu [Xi+23] und [Wan+23] für aktuelle Forschungstrends.

Die entwickelte Architektur implementiert Verteidigung in der Tiefe (Defense-in-Depth) durch mehrschichtige Sicherheitsmechanismen. Alle LLM-generierten Werkzeugaufrufe unterliegen einer Eingabevalidierung, die schädliche Aufrufe verhindert. Die Sandbox-Ausführung in isolierten Docker-Containern stellt sicher, dass potenziell gefährlicher Code nicht das Host-System kompromittieren kann. Ein Berechtigungssystem setzt das Prinzip der geringsten Rechte (Least-Privilege) durch und beschränkt Zugriffe auf das erforderliche Minimum. Kritische Operationen werden durch Audit-Protokollierung lückenlos protokolliert. Bei Deployment-kritischen Aktionen greift ein Human-in-the-Loop-Mechanismus, der menschliche Bestätigung erfordert.

Sicherheits-Audits zeigten 100 % Erfolgsrate bei der Abwehr von Prompt-Injection-Angriffen und unautorisiertem Dateizugriff.

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

Ein mehrdimensionales Metriken-Framework wurde entwickelt und validiert:

Funktional: Aufgabenerfolgsquote, Testbestehensquote, Lint-Bewertung, Review-Akzeptanz
Effizienz: Token-Verbrauch, API-Kosten, Laufzeit, Anzahl Werkzeugaufrufe
Robustheit: Fehlerbehebungsrate, Graceful Degradation bei Failures
Sicherheit: Policy-Verstöße, Sandbox-Ausbrüche, Zugriffsverweigerungen

Die 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 Kontextoptimierung
- **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 der Arbeit

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

1. **Referenzarchitektur:** Dokumentierte, wiederverwendbare Architektur für agentische SE-Workflows mit klaren Komponenten (Controller, Werkzeugregister, Gedächtnismanager, Sicherheitsschicht) und deren Interaktionen. Umfasst Design-Patterns, Schnittstellenspezifikationen 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-Richtlinien.
3. **Sicherheitsframework:** Konkrete Mechanismen für sichere Agentenausführung: Eingabevalidierung, Sandboxing, Permissions, Audit-Logging, Human-in-the-Loop. Getestet gegen Prompt-Injection und unautorisierten Zugriff.
4. **Evaluationsmethodik:** Mehrdimensionales Metrikenframework (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% Erfolgsrate, -40% Token-Kosten) gegenüber Baselines.
6. **Transferierbarkeit:** Architektur ist nicht auf SE beschränkt. Patterns (Sense-Plan-Act-Reflect, Werkzeugschnittstellen, Sicherheitsschicht) übertragbar auf andere Domänen (DevOps, Data Science, QA).

5.2.1 Wissenschaftlicher Beitrag

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

- Systematisierung agentischer SE-Architekturen (bisher meist ad-hoc Prototypen)
- Fokus auf Produktionsreife (Sicherheit, Kosten, Robustheit) statt nur Aufgabenerfolg
- 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. Praxiseinsätze 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.“ ([Jim+24])

5.3.2 Technische Limitierungen

- **Halluzinationen:** LLMs halluzinieren gelegentlich non-existente APIs oder fälschen Reasoning. Reflexion reduziert, eliminiert aber nicht.
- **Kontextfenster:** Trotz Optimierung stoßen 128 k-Token-Limits bei komplexen Codebasen an Grenzen. RAG/Chunking sind Workarounds, keine Lösungen.
- **Werkzeuglatenz:** Testläufe dauern Sekunden bis Minuten. Bei vielen Werkzeugaufrufen akkumuliert Latenz (Median: 45s für Test-Debugging).
- **Fehlerfortpflanzung:** Frühe Fehler (falsche Diagnose) propagieren durch iterative Loops. Ohne menschliche Aufsicht 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.*
>
> — Diskurs zur Arbeitsmarktentwicklung, vgl. Eloundou et al. [Elo+23]

Weitere ethische Dimensionen:

- **Bias-Verstetigung:** LLMs reproduzieren Biases aus Trainingsdaten. Code-Generierung kann diskriminierende Patterns fortführen.
- **Verantwortlichkeit:** Bei Agenten-generierten Bugs: Wer haftet? Entwickler? LLM-Anbieter? Unternehmen?
- **Übermäßiges Vertrauen:** Entwickler könnten kritisches Denken reduzieren und blind Agent-Outputs vertrauen – gefährlich bei sicherheitskritischen Systemen.

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
- **Mehrsprachenunterstützung:** Aktuell Python-fokussiert. Erweiterung auf Java, TypeScript, Go für breitere Anwendbarkeit
- **Optimiertes Kontextmanagement:** Hybrid-Strategien (RAG + Summarization + Code-Graph-Navigation) für 1 M+ LOC Codebasen
- **Fine-Tuning:** Domänenspezifisches Fein-Tuning auf SE-Tasks könnte Erfolgsraten bei kleineren/günstigeren Modellen verbessern
- **Integration menschlichen Feedbacks:** RLHF-ähnliche Ansätze für kontinuierliches Lernen aus Entwicklerkorrekturen

5.4.2 Mittel- bis langfristige Forschungsrichtungen

- **Multi-Agenten-Systeme:** Rollenbasierte Kollaboration (Architect, Coder, Reviewer, Tester) wie in MetaGPT. Könnte Spezialisierung und Parallelisierung verbessern.
- **Kontinuierliches Lernen:** Agents lernen aus Projekt-Historie, Team-Patterns, Codebasis-Konventionen. Episodisches Gedächtnis als Trainingsdaten-Quelle.
- **Formale Verifikation:** Integration formaler Methoden (Typprüfung, SMT-Solving, symbolische Ausführung) für sicherheitskritischen Code.
- **IDE-Integration:** Tiefe Integration in VS Code, IntelliJ als Co-Pilot++ mit direktem Arbeitsbereichszugriff und Echtzeitvorschlägen.
- **Hybride Mensch-Agent-Workflows:** Optimale Arbeitsteilung: Was automatisieren? Wo menschliche Expertise essential? Tooling für effiziente Delegation und Review.

- **Kosten-Nutzen-Optimierung:** Automatische Entscheidung wann Agent, wann Mensch basierend auf Aufgabenkomplexität, Deadline, Budgetbeschränkungen.

5.4.3 Offene Forschungsfragen

- **Vertrauenswürdigkeit:** Wie messen/garantieren wir Vertrauenswürdigkeit? Formale Spezifikationen für Agentenverhalten?
- **Emergentes Verhalten:** Bei komplexen Multi-Agenten-Systemen: Wie kontrollieren/verstehen wir emergentes Verhalten?
- **Langzeitgedächtnis:** 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? Domänenadaption für neue Codebasen?

5.5 Schlusswort

Die 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:

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

Die Technologie ist nicht „autonom genug“ für Vollautomatisierung, aber wertvoll als Erweiterungswerkzeug 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.

Konkrete Empfehlungen für Praktiker:

- Beginnen Sie mit klar abgegrenzten, gut getesteten Teil-Workflows (z. B. Lint-Fixes, kleine Refactorings) bevor Sie größere Automatisierungsebenen freischalten.
- Implementieren Sie schrittweise Human-in-the-Loop-Gates für kritische Aktionen und messen Sie kontinuierlich Metriken wie Review-Akzeptanz und Regressionen.
- Nutzen Sie Vektor-basierte Retrieval-Mechanismen für langfristige Projekte, um Kontextkosten zu reduzieren, und automatisieren Sie Summarisierungspipelines für alte Commits.
- Planen Sie regelmäßige Sicherheits-Audits und erweitern Sie Test-Suites um adversariale Prompt-Tests.

Die pragmatischen Schritte erleichtern den verantwortungsvollen Einsatz agentischer Systeme im Alltag und minimieren Risiken beim Übergang in produktive Umgebungen.

Literaturverzeichnis

[Cha22] Harrison Chase. *LangChain*. https://github.com/langchain-ai/langchain. 2022. (besucht am 05. 01. 2026).

[Deu23] Deutsche Forschungsgemeinschaft. *Umgang mit generativen Modellen zur Text- und Bilderstellung in der Wissenschaft und im Förderhandeln der DFG*. Stellungnahme des Präsidiums. 2023. URL: https://www.dfg.de/resource/blob/289674/230921-stellungnahme-praesidium-ki-ai.pdf (besucht am 05. 01. 2026).

[Elo+23] Tyna Eloundou u. a. „GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models“. In: *arXiv preprint arXiv:2303.10130* (2023).

[Hon+24] Sirui Hong u. a. „MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework“. In: *International Conference on Learning Representations (ICLR)*. 2024.

[Jim+24] Carlos E. Jimenez u. a. „SWE-bench: Can Language Models Resolve Real-World GitHub Issues?“ In: *International Conference on Learning Representations (ICLR)*. 2024.

[Kar23] Andrej Karpathy. *Intro to Large Language Models*. https://karpathy.ai/stateofgpt.pdf. 2023. (besucht am 05. 01. 2026).

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[Ope24] OpenAI. *GPT-4 Technical Report*. Techn. Ber. OpenAI, 2024. URL: https://arxiv.org/abs/2303.08774.

[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*. 2023.

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

[RF20] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020.

[Sch+23] Timo Schick u. a. „Toolformer: Language Models Can Teach Themselves to Use Tools". In: *arXiv preprint arXiv:2302.04761* (2023).

[Shi+23] Noah Shinn u. a. „Reflexion: Language Agents with Verbal Reinforcement Learning". In: *Advances in Neural Information Processing Systems*. 2023.

[Som15] Ian Sommerville. *Software Engineering*. 10th. 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*. 2017.

[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: *Advances in Neural Information Processing Systems*. 2022.

[Wen23] Lilian Weng. *Prompt Engineering*. https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/. 2023. (besucht am 05. 01. 2026).

[Wu+23] Qingyun Wu u. a. „AutoGen: Enabling Next-Gen LLM Applications". In: *arXiv preprint arXiv:2308.08155* (2023).

[Xi+23] Zhiheng Xi u. a. „The Rise and Potential of Large Language Model Based Agents: A Survey". In: *arXiv preprint arXiv:2309.07864* (2023).

[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: *International Conference on Learning Representations (ICLR)*. 2023.

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

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

Index

A Verzeichnis der KI-Nutzung

Erklärung zur Nutzung KI-basierter Werkzeuge

Die Erstellung der vorliegenden Arbeit wurde durch den Einsatz KI-basierter Werkzeuge unterstützt. Eine detaillierte tabellarische Übersicht der verwendeten Systeme sowie des jeweiligen Zwecks und Umfangs ihrer Nutzung findet sich in Tabelle A.1.

Alle von KI-Systemen generierten Vorschläge und Inhalte wurden von mir eigenständig geprüft, fachlich bewertet und bei Bedarf überarbeitet oder verworfen. Die Verantwortung für die Auswahl, inhaltliche Einordnung, Interpretation sowie die endgültige Fassung des Textes liegt vollständig bei mir.

Grundlage für den Umgang mit den Werkzeugen bilden die aktuellen Empfehlungen der Deutschen Forschungsgemeinschaft (DFG) zu KI in wissenschaftlichen Arbeiten [Deu23].

Declaration on the Use of AI-based Tools

The preparation of this thesis was supported by the use of AI-based tools. A detailed tabular overview of the systems used, as well as the respective purpose and scope of their use, is provided in Table A.1.

All suggestions and content generated by AI systems were independently reviewed, professionally evaluated, and revised or rejected by me where necessary. I assume full responsibility for the selection, categorization, interpretation, and the final version of the text.

The handling of these tools follows the current recommendations of the German Research Foundation (DFG) for scientific work [Deu23].

Tabelle A.1 listet alle in dieser Arbeit verwendeten KI-Werkzeuge auf.

Tabelle A.1: Verzeichnis der in dieser Arbeit genutzten KI-Werkzeuge

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
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

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 5. Januar 2026 Maria Musterfrau