Agentic Software Development

Enterprise Architecture with AI Agents

Praxisbeispiele mit Java, Spring Boot und Domain-Driven-Design

Multi-Agent-System · Shared Knowledge Stores · DDD-Integration · AI Guardrails

Martin Janda

IT-Architekt | AI & Enterprise Integration

www.janda.io

Version 1.2 · März 2026

Inhalt

Abbildungsverzeichnis

Management Summary

Die Softwareentwicklung steht vor einem Paradigmenwechsel: KI-gestützte Agenten übernehmen nicht nur die Code-Generierung, sondern orchestrieren den gesamten Entwicklungslebenszyklus – von der Anforderungsanalyse über die Implementierung bis zum Deployment. Dieses Dokument beschreibt eine produktionsreife Enterprise-Architektur für den Einsatz solcher Agentensysteme in regulierten Umgebungen.

Kernaussagen

Multi-Agent statt Monolith: Sieben spezialisierte Agenten (Architektur, Planung, Requirements, Entwicklung, Testing, Review, Deployment) arbeiten koordiniert in einer Hub-and-Spoke-Architektur. Jeder Agent hat klar definierte Verantwortlichkeiten und eingeschränkte Werkzeugzugriffe – vergleichbar mit einem erfahrenen Entwicklungsteam.

Governance by Design: Eine sechsstufige Guardrails-Pipeline (Syntax, Style, Security, Domain-Compliance, Tests, Confidence Scoring) validiert jeden generierten Code-Artefakt automatisch. Kein Code gelangt ohne bestandene Validierung in die Codebasis. Dies adressiert das Kernrisiko von KI-Systemen: Halluzinationen und unkontrollierte Seiteneffekte.

Enterprise-tauglich: Die Architektur berücksichtigt Domain-Driven Design, rollenbasierte Zugriffskontrolle, ein fünfschichtiges Memory-Modell, formale Architekturentscheidungen (ADRs) und ein transparentes Kostenmodell mit Token- und Execution-Budgets. Sie ist damit auch für regulierte Umgebungen (Behörden, Finanzwesen) geeignet.

Wirtschaftlicher Nutzen

Für ein typisches Feature (z. B. einen Payment-Service mit DDD, Event-Driven Architecture und Kubernetes-Deployment) reduziert der Agenten-Einsatz den manuellen Entwickleraufwand um 70–80 %. Die API-Kosten liegen bei €15–30 pro Feature – einem Bruchteil der eingesparten Personalkosten. Die Time-to-Feature sinkt von 1–2 Wochen auf 1–2 Tage, bei gleichzeitig höherer Testabdeckung (>80 % vs. oft <60 % unter Zeitdruck).

Zielgruppe

Dieses Dokument richtet sich an IT-Architekten, technische Projektleiter und Entwicklungsteams, die KI-Agenten systematisch in ihren Software Development Lifecycle integrieren möchten. Alle Beispiele verwenden Java 21, Spring Boot und bewährte Enterprise-Patterns – die Prinzipien sind jedoch technologieunabhängig übertragbar.

TEIL I: GRUNDLAGEN & ARCHITEKTUR

1. Warum KI-Agenten in der Softwareentwicklung?

Die Softwareentwicklung steht vor einem fundamentalen Paradigmenwechsel. Während bisherige Automatisierungsansätze – von CI/CD-Pipelines über Code-Generatoren bis hin zu IDE-Plugins – stets auf klar definierte, deterministisch ablaufende Aufgaben beschränkt waren, eröffnen KI-gestützte Agenten eine völlig neue Dimension: sie können kontextabhängig entscheiden, iterativ arbeiten und komplexe Aufgabenketten selbstständig orchestrieren.

Claude Code, Anthropics agentisches CLI-Tool, geht über die reine Code-Generierung weit hinaus. Es ermöglicht ein vollständiges Multi-Agent-System, bei dem spezialisierte Subagenten autonom und koordiniert arbeiten – vergleichbar mit einem erfahrenen Entwicklungsteam, bei dem jedes Mitglied seine Expertise einbringt. Die Agenten lesen bestehenden Code, analysieren Architekturen, schreiben Tests, führen Sicherheitsreviews durch und deployen Anwendungen – alles innerhalb eines kohärenten Workflows.

Dieses Dokument beschreibt im Detail, wie Enterprise-Teams Claude Code Agenten in ihren Software Development Lifecycle (SDLC) integrieren können. Der besondere Fokus liegt auf drei Kernaspekten: Erstens einem geteilten Wissensstand zwischen Agenten, damit Erkenntnisse aus der Architekturanalyse nahtlos in die Implementierung einfließen. Zweitens einer fachlichen Modellierung nach DDD-Prinzipien, die sicherstellt, dass der generierte Code nicht nur technisch korrekt, sondern auch fachlich präzise ist. Drittens einem robusten AI Risk Framework mit Guardrails, das Halluzinationen erkennt und verhindert, bevor fehlerhafter Code in die Codebasis gelangt.

Das Orchestrator-Prinzip

Das Herzstück der Claude-Code-Agentenarchitektur ist das Orchestrator-Prinzip. Die Claude-Code-Hauptsitzung fungiert als zentrale Steuerungsinstanz – vergleichbar mit einem Technical Lead, der Aufgaben verteilt, Fortschritte überwacht und Ergebnisse zusammenführt.

Über das Task-Tool werden spezialisierte Subagenten gestartet, wobei jeder Agent sein eigenes Kontextfenster besitzt. Dies ist ein entscheidender architektureller Vorteil: Jeder Subagent arbeitet in einem isolierten Kontext, was Interferenzen zwischen parallelen Aufgaben verhindert. Gleichzeitig können Agenten über den Shared Knowledge Store Informationen austauschen, ohne ihre Isolation zu durchbrechen.

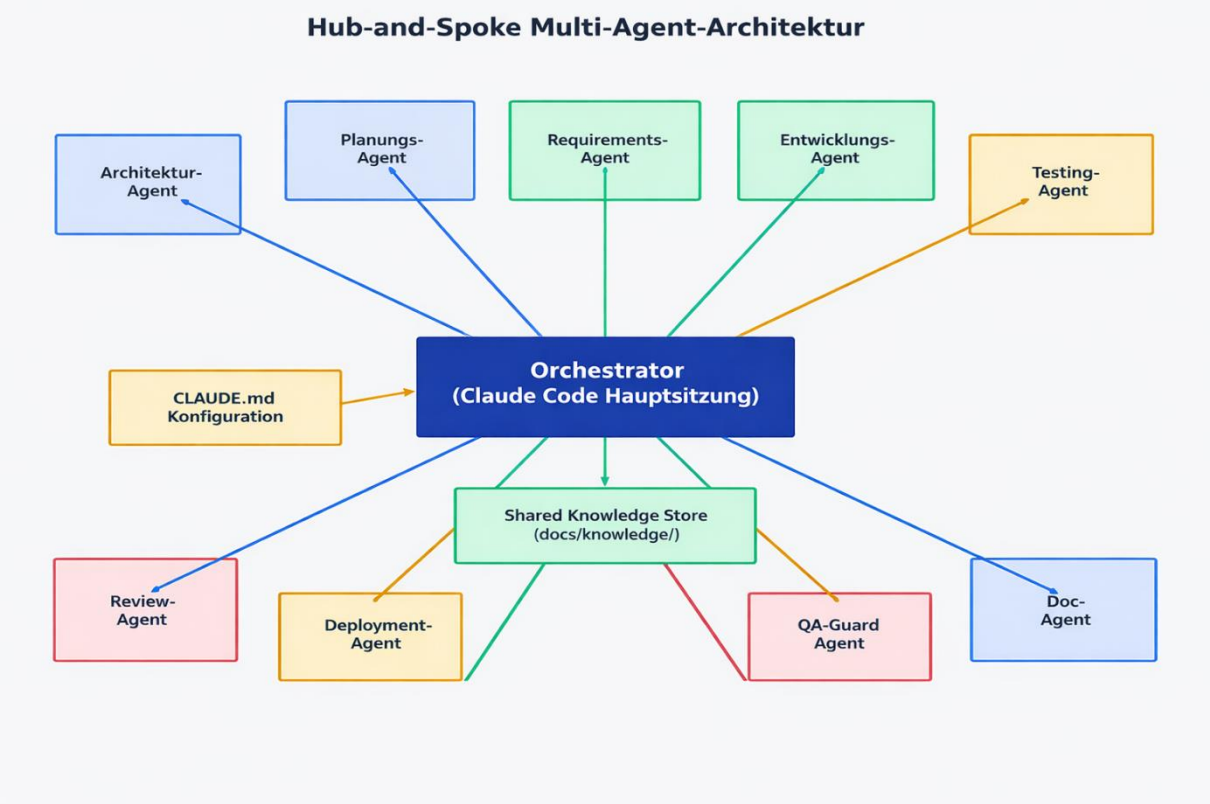


Abbildung 1: Hub-and-Spoke Multi-Agent-Architektur mit zentralem Orchestrator

Kernprinzipien des Orchestrator-Modells

Prinzip	Beschreibung
Separation of Concerns	Jeder Agent hat eine klar definierte Rolle und einen begrenzten Werkzeugzugriff.
Parallele Ausführung	Unabhängige Agenten laufen gleichzeitig. Testing, Dokumentation und Review können parallel stattfinden.
Autonome Operation	Subagenten arbeiten eigenständig innerhalb ihres definierten Rahmens.
Ergebnis-Aggregation	Der Orchestrator sammelt und synthetisiert die Outputs aller Subagenten.
Konfigurierbarkeit	Benutzerdefinierte Agenten und Regeln werden über CLAUDE.md deklarativ konfiguriert.
Wiederaufnahme	Agenten können über ihre Agent-ID fortgesetzt werden.
Shared State	Ein gemeinsamer Wissensstand über den Shared Knowledge Store ermöglicht Zusammenarbeit.
Guardrails	Ein AI Risk Framework mit Halluzinationserkennung sichert die Codequalität.

2. Architektonische Prinzipien

Jedes Enterprise-Architekturdokument basiert auf einem Set formaler Prinzipien, die als Leitplanken für alle Designentscheidungen dienen. Die folgenden sechs Prinzipien bilden das Fundament des Claude Code Agentensystems und werden in den nachfolgenden Kapiteln durchgehend referenziert.

Prinzip	Kurzformel	Beschreibung
AP-1: Agent Specialization	Ein Agent, eine Rolle	Jeder Agent hat genau eine klar definierte Verantwortlichkeit. Kein Agent vereint Entwicklung und Review. Spezialisierung führt zu höherer Qualität und ermöglicht gezielte Modellauswahl (opus für Architektur, haiku für Formatierung).
AP-2: Deterministic Execution	Reproduzierbar und nachvollziehbar	Trotz nicht-deterministischer LLM-Outputs sorgen Task-Graphen, Stop-Conditions und Execution Budgets für vorhersagbare Gesamtabläufe. Jeder Lauf ist über Audit-Logs vollständig nachvollziehbar.
AP-3: Governance by Design	Compliance eingebaut, nicht nachgerüstet	Sicherheit, Domain-Compliance und Qualitätssicherung sind keine optionalen Add-ons, sondern integraler Bestandteil jeder Pipeline-Stufe – durchgesetzt durch Hooks und Guardrails.
AP-4: Isolation by Workspace	Kein Agent verändert unkontrolliert	Jeder Agent arbeitet in einem isolierten Workspace (Git Worktree). Änderungen werden erst nach erfolgreicher Validierung in den Haupt-Branch überführt. Parallele Agenten können sich nicht gegenseitig beeinflussen.
AP-5: Policy-Driven Development	Regeln statt Hoffnung	Projektstandards, DDD-Regeln, Coding Conventions und Architekturvorgaben werden deklarativ in CLAUDE.md definiert – nicht als Prosa-Dokumentation, sondern als maschinenlesbare Policies.
AP-6: Human-in-the-Loop	KI assistiert, Mensch entscheidet	Kritische Entscheidungen (Architektur-ADRs, Security-Findings, Deployment-Freigaben) erfordern immer menschliche Bestätigung. KI-Agenten sind Werkzeuge, keine autonomen Entscheider.

Diese Prinzipien stehen nicht isoliert, sondern bedingen sich gegenseitig: Agent Specialization (AP-1) ermöglicht erst Isolation by Workspace (AP-4), weil spezialisierte Agenten klare Workspace-Grenzen haben. Governance by Design (AP-3) setzt Policy-Driven Development (AP-5) voraus, weil Hooks nur durchsetzen können, was als Policy definiert ist. Und Human-in-the-Loop (AP-6) ist das Sicherheitsnetz für alle Fälle, in denen Deterministic Execution (AP-2) an seine Grenzen stößt.

Zuordnung zu Architekturkonzepten

Prinzip	Kapitelreferenzen	Durchsetzung
AP-1: Agent Specialization	Kap. 5 (Agententypen), Kap. 12 (Berechtigungsmodell)	CLAUDE.md Agentendefinitionen + Tool-Restriktionen
AP-2: Deterministic Execution	Kap. 7 (Execution Model), Kap. 18 (ADR-4)	Task-Graph, max_turns, Execution Budgets
AP-3: Governance by Design	Kap. 13 (Guardrails), Kap. 17 (Hooks)	PreToolUse/PostToolUse Hooks, Validierungs-Pipeline
AP-4: Isolation by Workspace	Kap. 15 (Workflows), Kap. 18 (ADR-2)	Git Worktrees, isolation="worktree" Parameter
AP-5: Policy-Driven Development	Kap. 4 (CLAUDE.md), Kap. 12 (DDD)	CLAUDE.md als Single Source of Truth
AP-6: Human-in-the-Loop	Kap. 9 (Failure Handling), Kap. 14 (Security)	Confidence Score Eskalation, Review Gates

3. Die Agentenarchitektur im Überblick

3.1 Systemkontext des agentischen Entwicklungssystems

Bevor die interne Architektur des agentischen Entwicklungssystems betrachtet wird, ist es hilfreich, den Systemkontext zu verstehen.

Das agentische Entwicklungssystem ist nicht isoliert, sondern integriert sich in die bestehende Enterprise-Toolchain.

Typische Integrationspunkte sind:

- Versionsverwaltung (Git)
- CI/CD-Pipelines
- Knowledge Stores und Dokumentation
- Container-Runtime und Deploymentplattformen

Der Orchestrator bildet dabei die zentrale Steuerungseinheit, während spezialisierte Agenten Entwicklungsaufgaben automatisiert ausführen.

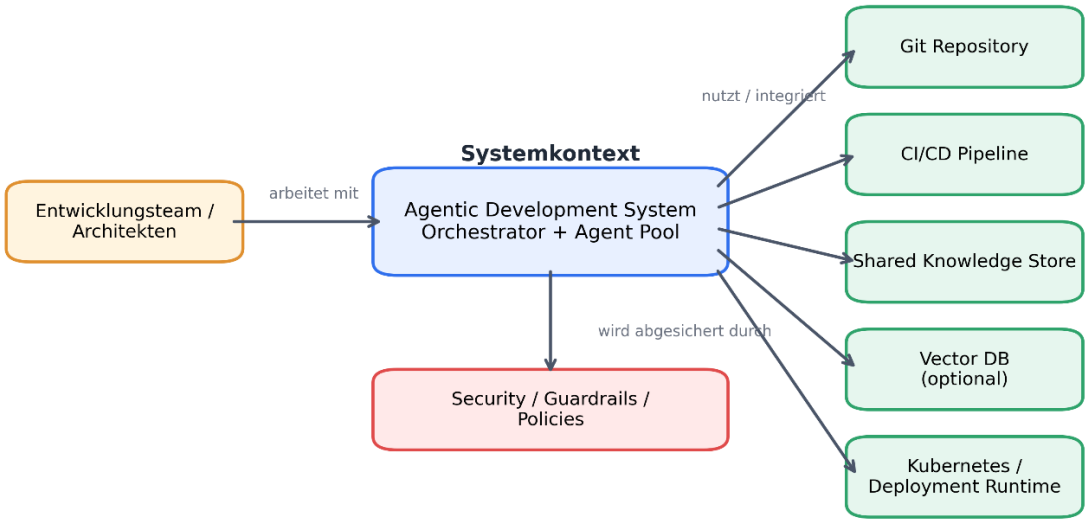


Abbildung 2: Systemkontext

3.2 Referenzarchitektur des Agentensystems

Die Architektur folgt einem erweiterten Hub-and-Spoke-Modell. Im Zentrum steht der Orchestrator, der Arbeit an spezialisierte Subagenten verteilt, deren Ergebnisse koordiniert und einen gemeinsamen Wissensstand (Shared Knowledge Store) verwaltet. Dieses Modell bietet entscheidende Vorteile gegenüber einem monolithischen Agenten: Spezialisierung führt zu höherer Qualität, Parallelisierung beschleunigt den Gesamtprozess, und Isolation verhindert, dass ein fehlerhafter Agent die gesamte Pipeline beeinträchtigt.

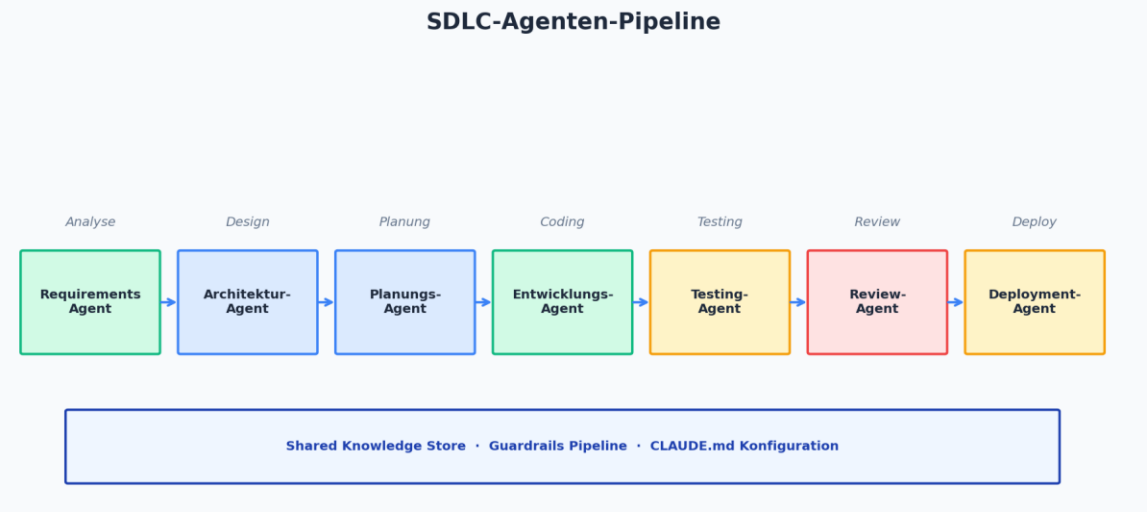


Abbildung 3: SDLC-Agenten-Pipeline – von der Anforderungsanalyse bis zum Deployment

3.3 Agentenübersicht

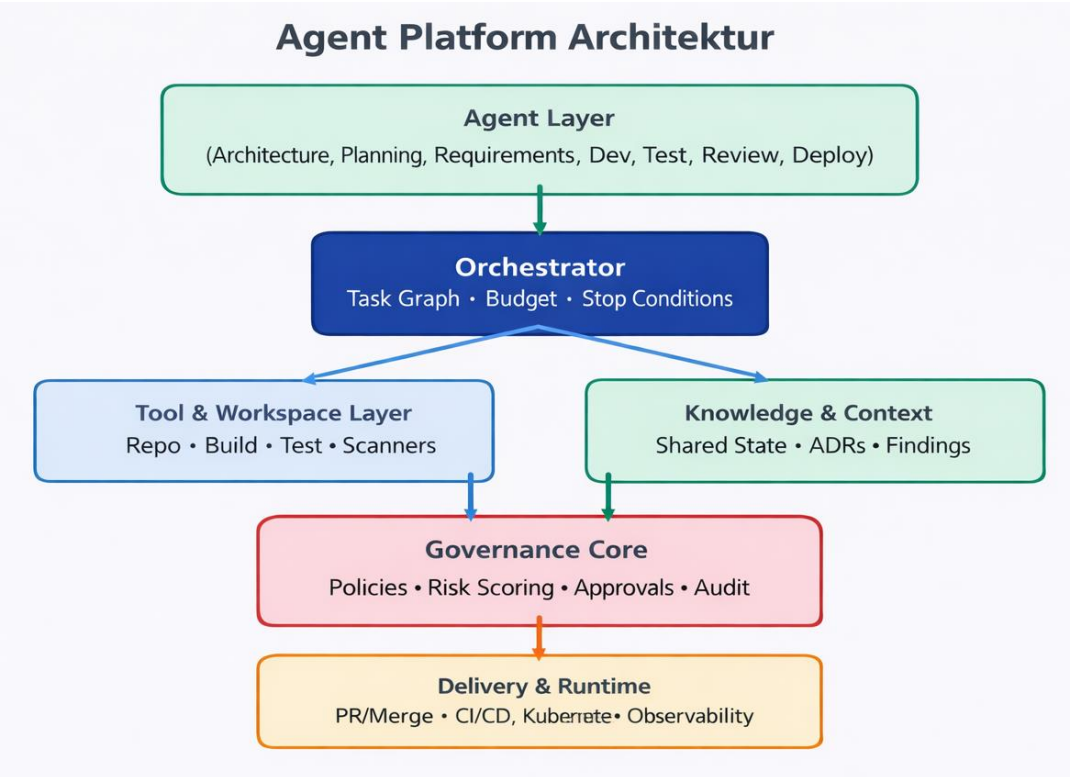


Abbildung 4: Referenzarchitektur eines agentischen Entwicklungssystems im Enterprise-Kontext

Der Agent Layer besteht aus spezialisierten Rollen (Architektur, Planung, Requirements, Entwicklung, Testing, Review, Deployment), die jeweils klar begrenzte Verantwortlichkeiten besitzen.

Der Orchestrator übersetzt Ziele in einen Task-Graph, steuert Zustandsübergänge, Budgets und Stop-Conditions und sorgt für deterministische Abläufe.

Der Tool & Workspace Layer kapselt alle mutierenden Aktionen über Tool-Adapter und isolierte Workspaces, um unkontrollierte Seiteneffekte zu verhindern.

Der Governance Core (Execution Contracts, Risk Scoring, Policies, Approvals, Ledger/Audit) stellt sicher, dass Änderungen nur innerhalb definierter Regeln stattfinden.

Delivery & Runtime umfasst PR/Merge, Signierung/SBOM, CI/CD-Gates, Kubernetes Admission Policies sowie Observability.

3.4 Runtime-Architektur

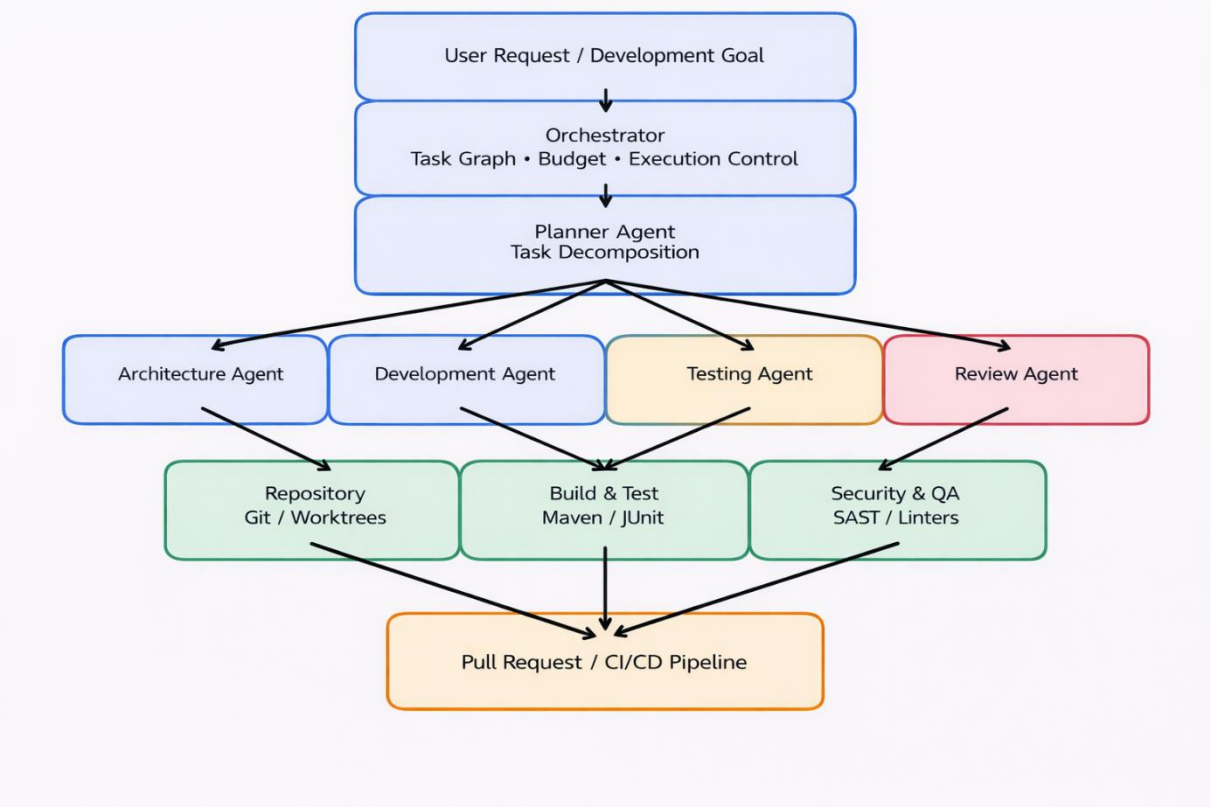


Abbildung 5: Runtime-Architektur eines agentischen Entwicklungssystems

Die Runtime-Architektur beschreibt den Ablauf eines Entwicklungsauftrags während der tatsächlichen Ausführung. Ein Entwicklungsziel wird zunächst vom Orchestrator entgegengenommen, der Zustände verwaltet, Budgets kontrolliert und Stop-Conditions definiert. Ein Planner-Agent zerlegt dieses Ziel anschließend in konkrete Aufgaben und delegiert sie an spezialisierte Agenten im Agent Pool.

Agent	Rolle	Werkzeugzugriff
Orchestrator	Verteilt Aufgaben, koordiniert Phasen, verwaltet Shared State	Alle Tools inkl. Task
Architektur-Agent	Analysiert Systemdesign, erstellt ADRs, bewertet Patterns	Read, Glob, Grep, LSP
Planungs-Agent	Erstellt detaillierte Implementierungspläne mit Meilensteinen	Read, Glob, Grep, Write
Requirements-Agent	Sammelt, validiert und trackt Anforderungen	Read, Glob, Grep, WebFetch
Entwicklungs-Agent	Implementiert Features nach Projektstandards und DDD	Read, Write, Edit, Bash, LSP
Testing-Agent	Erstellt und führt Unit-, Integrations- und E2E-Tests aus	Read, Write, Edit, Bash, Glob

Review-Agent	Prüft Code auf Qualität, Sicherheit und Domain-Compliance	Read, Glob, Grep, LSP
Deployment-Agent	Verwaltet IaC, führt Deployments mit K8s und Terraform aus	Read, Write, Edit, Bash

3.5 Task-Tool Parameter

Parameter	Pflicht	Typ	Beschreibung
subagent_type	Ja	string	Welcher Agententyp gestartet wird (z.B. "dev-agent")
prompt	Ja	string	Detaillierte Aufgabenanweisungen mit Kontext und Erwartungen
description	Ja	string	Kurze 3–5 Wort Zusammenfassung für das Logging
model	Nein	enum	sonnet (Standard), opus für komplexe Aufgaben, haiku für schnelle Tasks
max_turns	Nein	integer	Maximale API-Roundtrips bevor der Agent automatisch stoppt
run_in_background	Nein	boolean	Asynchrone Ausführung – gibt sofort die Agent-ID zurück
isolation	Nein	enum	"worktree" für isolierte Git-Worktree-Kopie des Repositories
resume	Nein	string	Agent-ID zum Fortsetzen einer unterbrochenen Sitzung

4. Konfiguration mit CLAUDE.md

CLAUDE.md ist die zentrale Konfigurationsdatei für das Agenten-Verhalten, Team-Standards und Projektregeln. Sie funktioniert wie eine Kombination aus .editorconfig, ESLint-Konfiguration und Architektur-Dokumentation – nur für KI-Agenten. Die Datei ist Klartext im Markdown-Format und wird von jedem Agenten beim Context Build (Lifecycle-Phase 2) automatisch geladen. Damit ist CLAUDE.md die operative Umsetzung des Prinzips AP-5 (Policy-Driven Development): Was hier steht, wird durchgesetzt.

Konfigurationsebenen

CLAUDE.md folgt einem dreistufigen Vererbungsmodell. Höhere Ebenen können durch niedrigere überschrieben werden, ähnlich wie CSS-Spezifizität:

Geltungsbereich	Ort	Zweck
Global (Nutzer)	~/.claude/CLAUDE.md	Persönliche Defaults für alle Projekte: bevorzugter Stil, Standardsprache, globale Regeln
Projekt (Team)	./CLAUDE.md (Repo-Root)	Team-gemeinsame Konfigurationen: Agentendefinitionen, Architekturregeln, DDD-Glossar, CI/CD-Standards
Lokal (Entwickler)	./CLAUDE.local.md	Persönliche Overrides (nicht im Git): IDE-Pfade, lokale DB-URLs, experimentelle Flags

Die Vererbung funktioniert so: Global definiert Baselines, Projekt überschreibt für das Team, Lokal überschreibt für den einzelnen Entwickler. Konflikte werden nach dem Last-Wins-Prinzip aufgelöst – die spezifischste Konfiguration gewinnt.

4.1 Aufbau und Sektionen

Eine vollständige CLAUDE.md besteht aus mehreren klar abgegrenzten Sektionen. Jede Sektion adressiert einen spezifischen Aspekt der Agenten-Steuerung:

Sektion	Inhalt	Wirkung
Agentendefinitionen	Welche Custom Agents existieren, mit welchen Tools und Rollen	Steuert AP-1 (Agent Specialization) und das Berechtigungsmodell
Projektstandards	Java-Version, Style Guide, Framework-Versionen, Package-Struktur	Agenten generieren Code konform zu diesen Standards
DDD-Regeln	Bounded Contexts, Glossar-Pfad, Event-Naming, Context Map	Domain-Hooks prüfen gegen diese Regeln (AP-3)
Architekturregeln	Erlaubte Patterns, verbotene Anti-Patterns, Schichtentrennung	Review-Agent nutzt diese als Prüfkatalog
Cost Controls	Default-Modell, max_turns, Budget-Limits, Eskalationsregeln	Steuert Execution Budget (Kap. 7) und Token Budget (Kap. 14)
Hooks & MCP	PreToolUse/PostToolUse Hooks, MCP-Server-Referenzen	Governance by Design (AP-3) durch automatische Validierung
Verbotene Aktionen	Dateien/Pfade die nie geändert werden dürfen, gesperrte Kommandos	Sicherheitsleitplanken über das Tool-Whitelisting hinaus

4.2 Vollständiges Enterprise-Beispiel

Das folgende Beispiel zeigt eine produktionsreife CLAUDE.md für ein Banking-Projekt. Jede Sektion ist kommentiert, um die Wirkung auf das Agenten-Verhalten zu erklären:

```
# ============================================================
# Projekt: Banking-Plattform (Export Manager)
# Team: Backend Engineering, BAFA Modernisierung
# ============================================================

# --- Agentendefinitionen ---
# Jeder Agent hat: Name, Rolle, erlaubte Tools, Beschränkungen
# Dies setzt AP-1 (Agent Specialization) operativ um.

## Agenten
- architecture-agent: Analysiert Systemdesign, erstellt ADRs.
  Tools: Read, Glob, Grep, LSP, WebFetch
  Modell: opus (für maximale Analysetiefe)
  Einschränkung: Kein Schreibzugriff auf Code

- dev-agent: Implementiert Features nach Projektstandards.
  Tools: Read, Write, Edit, Glob, Grep, Bash, LSP
  Modell: sonnet (Kosten-/Qualitätsbalance)
  Regel: MUSS bestehenden Code lesen BEVOR Änderungen vorgenommen werden

- test-agent: Schreibt und führt Tests aus.
  Tools: Read, Write, Edit, Bash, Glob, Grep
  Regel: Iteriert bis 100% Bestehensrate
  Einschränkung: Darf nur Dateien in src/test/ ändern

- review-agent: Prüft Code auf Qualität und Sicherheit.
  Tools: Read, Glob, Grep, LSP
  Modell: opus (Security-relevante Analyse)
  Einschränkung: Kein Schreibzugriff

- deploy-agent: Erstellt IaC-Artefakte.
  Tools: Read, Write, Edit, Bash
  Einschränkung: Darf nur Dateien in deploy/ und k8s/ ändern

# --- Projektstandards ---
# Agenten generieren Code EXAKT nach diesen Vorgaben.

## Technologie-Stack
- Java 21 LTS, Google Java Style Guide
- Spring Boot 3.4, Maven Multi-Module
- JUnit 5, Testcontainers, WireMock
- Angular 21 (Frontend), NX Monorepo
- Camunda 8 (BPMN Workflow Engine)

## Package-Struktur
- com.company.{bounded-context}.domain       # Entities, Value Objects, Events
- com.company.{bounded-context}.application  # Services, Use Cases
- com.company.{bounded-context}.adapter.in   # REST Controller, GraphQL
- com.company.{bounded-context}.adapter.out  # JPA Repos, Kafka, External APIs
- com.company.{bounded-context}.config       # Spring Configuration

## Code-Konventionen
- Constructor Injection (kein @Autowired auf Feldern)
- @Slf4j für Logging (kein System.out)
- Records für DTOs und Value Objects
- Sealed Interfaces für Strategy Patterns
- @Transactional nur auf Service-Layer

# --- DDD-Regeln ---
# Definiert fachliche Grenzen, die Agenten einhalten MÜSSEN.

## Bounded Contexts
- ausfuhr-context: Ausfuhrgenehmigungen, KN-Codes, TARIC
- pruefung-context: Prüfverfahren, Vier-Augen-Prinzip
- stammdaten-context: Antragsteller, Firmen, Adressen
- dokument-context: Dokumenten-Upload, Signierung

## Ubiquitous Language
- Glossar: docs/domain/glossary.md (MUSS vor jeder Implementierung gelesen werden)
- Context Map: docs/domain/context-map.json
- Events folgen: {Aggregate}{Verb}Event (z.B. AntragEingereichtEvent)

## Verbotene Übergriffe
- ausfuhr-context darf NICHT direkt auf stammdaten-context zugreifen
- Kommunikation zwischen Contexts NUR über Domain Events (Kafka)
- Keine JPA-Beziehungen über Context-Grenzen hinweg

# --- Cost Controls ---
# Verhindert unkontrollierten API-Verbrauch (siehe Kap. 14).
```

```
## Budget-Limits
- DEFAULT_MODEL: sonnet
- MAX_TURNS_DEFAULT: 15
- MAX_TURNS_REVIEW: 8
- SPRINT_BUDGET_USD: 200
- FEATURE_BUDGET_USD: 50
- ALERT_THRESHOLD: 0.8

## Modell-Eskalation
- opus NUR für: Architektur-ADRs, Security Reviews, Halluzinationsprüfung
- haiku NUR für: Code-Formatierung, einfache Umbenennung, Docs ohne Fachlogik

# --- Hooks ---
# Automatische Validierung bei jeder Tool-Nutzung (AP-3).

## PreToolUse
- Pfad-Whitelist: Schreibzugriffe nur auf src/, test/, docs/, deploy/
- Gesperrte Pfade: .env, secrets/, credentials.properties, CLAUDE.md
- Gesperrte Kommandos: rm -rf, DROP TABLE, curl (ohne Whitelist)

## PostToolUse
- checkstyle: Google Java Style (nach jedem Write/Edit)
- domain-compliance: Bounded Context Check (nach jedem Write in src/)
- secret-scanner: Credential-Erkennung (nach jedem Write/Edit)

# --- MCP-Server ---
# Externe Tool-Integrationen (Jira, DB, Confluence).

## Verfügbare Server
- jira-server: Sprint-Tickets, User Stories, Akzeptanzkriterien
- postgres-server: Datenbank-Schema-Abfragen (readonly!)
- confluence-server: Architektur-Dokumentation lesen

# --- Verbotene Aktionen ---
# Absolute Grenzen, die KEIN Agent überschreiten darf.

## Niemals
- Produktionsdatenbanken ändern oder löschen
- Credentials oder Secrets in Code oder Logs schreiben
- Dependencies ohne OWASP-Check hinzufügen
- Direkt auf main/master pushen (immer Feature-Branch + PR)
- CLAUDE.md selbst ändern
```

4.3 CLAUDE.md im Team-Workflow

CLAUDE.md ist eine Datei im Repository und unterliegt damit denselben Prozessen wie jeder andere Code: Sie wird versioniert, reviewed und über Pull Requests geändert. Das bedeutet, dass Änderungen an Agenten-Konfigurationen denselben Quality Gates unterliegen wie Codeänderungen – ein wichtiger Aspekt für Audit-Compliance.

In der Praxis empfiehlt sich folgendes Vorgehen: Das Team definiert die initiale CLAUDE.md gemeinsam in einem Architecture Workshop. Der Architektur-Agent wird als Erster konfiguriert, da seine Findings die Basis für alle anderen Agenten bilden. Anschließend werden die übrigen Agenten schrittweise hinzugefügt und in einem Pilotfeature getestet. Erkenntnisse aus dem Piloten fließen als Änderungen an CLAUDE.md zurück – der typische Feedback-Loop dauert 2–3 Sprints, bis die Konfiguration stabil ist.

> **Hinweis:** CLAUDE.md ist das mächtigste Steuerungsinstrument im Agentensystem. Eine gut gepflegte Konfiguration macht den Unterschied zwischen einem nützlichen Werkzeug und einer unkontrollierbaren Black Box. Investieren Sie Zeit in die initiale Konfiguration – es zahlt sich in jedem Sprint mehrfach aus.

5. Eingebaute Agententypen und Modellauswahl

Claude Code bietet vorkonfigurierte Agententypen, die auf die häufigsten SDLC-Aufgaben zugeschnitten sind. Zusätzlich können über CLAUDE.md eigene Agententypen definiert werden.

Agententypen und ihre SDLC-Phasen

Agententyp	Zweck	SDLC-Phase
Bash	Kommandoausführung, Git-Ops, Build-Automatisierung	Build / CI/CD
Explore	Codebase-Erkundung, Abhängigkeitsanalyse	Beliebig
Plan	Implementierungsstrategie mit Meilensteinen	Architektur
general-purpose	Mehrstufige Recherche und Analyse	Beliebig
implementation-planner	Erstellt Tracking-Dokumente und Aufgabenlisten	Planung
implementation-executor	Hohe Komplexität, mehrstufige Implementierung	Entwicklung
test-executor	Tests erstellen, ausführen und iterieren	Testing
qa-guard	Pre-Commit-Hooks, automatische Korrekturen	QA
doc-researcher	Dokumentation durchsuchen und analysieren	Requirements
doc-agent	Dokumentation erstellen und pflegen	Dokumentation

Modellauswahl-Strategie

Die Wahl des richtigen Modells hat erheblichen Einfluss auf Qualität, Geschwindigkeit und Kosten. Als Faustregel gilt: opus für Entscheidungen, bei denen Fehler teuer wären (Architektur, Security), sonnet als Arbeitspferd für den Großteil der Entwicklung, und haiku für schnelle, unkritische Aufgaben.

TEIL II: KERNKONZEPTE & LAUFZEITMODELL

6. Agent Lifecycle

> **Hinweis:** Abgrenzung: Dieses Kapitel beschreibt den Lebenszyklus eines einzelnen Agenten. Das Execution Model (Kap. 7) beschreibt die Orchestrierung mehrerer Agenten. Die Workflow-Patterns (Kap. 15) zeigen die konkreten Aufruf-Beispiele.

Jeder Claude Code Agent durchläuft einen definierten Lebenszyklus von der Erstellung bis zur Terminierung. Das Verständnis dieses Lifecycles ist entscheidend für die Konfiguration von Timeouts, Budget-Limits und Retry-Strategien. Der Lifecycle besteht aus acht klar abgegrenzten Phasen:

#	Phase	Beschreibung
1	Spawn	Der Orchestrator erzeugt den Subagenten über das Task-Tool mit Typ, Prompt, Modell und optionalen Parametern (max_turns, isolation, run_in_background). Der Agent erhält eine eindeutige Agent-ID.
2	Context Build	Der Agent lädt seinen Arbeitskontext: CLAUDE.md-Konfiguration, relevante Dateien aus dem Repository, Einträge aus dem Shared Knowledge Store und den Prompt des Orchestrators. Diese Phase bestimmt maßgeblich den Token-Verbrauch.
3	Planning	Der Agent analysiert die Aufgabe und erstellt einen internen Ausführungsplan. Bei komplexen Tasks wird dieser Plan explizit als Implementierungs-Tracker persistiert. Der Planungsschritt ist bei implementation-planner Agenten die Hauptausgabe.
4	Execution	Die Kernarbeit: Der Agent führt die geplanten Schritte aus, ruft Tools auf und generiert Artefakte. Jeder API-Roundtrip zählt als ein "Turn" gegen das max_turns-Limit. Die Execution-Phase kann mehrere Tool-Aufrufe pro Turn umfassen.
5	Tool Interaction	Innerhalb der Execution-Phase interagiert der Agent mit seinem definierten Werkzeugset. Jede Tool-Nutzung wird durch PreToolUse- und PostToolUse-Hooks validiert. Werkzeugzugriffe außerhalb der Konfiguration werden blockiert.
6	Validation	Nach Abschluss der Execution durchläuft der Output die Guardrails-Pipeline: Syntax, Style, Security, Domain-Compliance, Tests und Confidence Scoring. Bei Fehlern wird der Agent in die Execution-Phase zurückgesetzt (Retry-Loop).
7	Termination	Der Agent beendet sich durch: (a) erfolgreichen Abschluss, (b) Erreichen von max_turns, (c) Budget-Erschöpfung, (d) explizite Stop-Condition, oder (e) Fehler-Eskalation. Das Ergebnis wird an den Orchestrator zurückgegeben.
8	Memory Update	Erkenntnisse, Findings und Entscheidungen werden in den Shared Knowledge Store geschrieben. Dieser Schritt ist essentiell für die Wissensübergabe an nachfolgende Agenten im Workflow.

Java-Beispiel: Agent Lifecycle Manager

```java
// === Agent Lifecycle State Machine ===
public class AgentLifecycle {

    public enum Phase {
        SPAWNED, CONTEXT_BUILDING, PLANNING, EXECUTING,
        TOOL_INTERACTION, VALIDATING, TERMINATED, MEMORY_UPDATING;

        public Set<Phase> allowedTransitions() {
            return switch (this) {
                case SPAWNED -> Set.of(CONTEXT_BUILDING);
                case CONTEXT_BUILDING -> Set.of(PLANNING);
                case PLANNING -> Set.of(EXECUTING, TERMINATED);
                case EXECUTING -> Set.of(TOOL_INTERACTION, VALIDATING, TERMINATED);
                case TOOL_INTERACTION -> Set.of(EXECUTING);
                case VALIDATING -> Set.of(EXECUTING, MEMORY_UPDATING, TERMINATED);
```

```
                case MEMORY_UPDATING -> Set.of(TERMINATED);
                case TERMINATED -> Set.of();
            };
        }
    }

    public record AgentState(
            String agentId,
            String agentType,
            Phase currentPhase,
            int turnsUsed,
            int maxTurns,
            Instant spawnedAt,
            @Nullable Instant terminatedAt,
            TerminationReason terminationReason,
            List<PhaseTransition> transitionLog
    ) {
        public boolean canTransitionTo(Phase target) {
            return currentPhase.allowedTransitions().contains(target);
        }

        public Duration elapsed() {
            Instant end = terminatedAt != null ? terminatedAt : Instant.now();
            return Duration.between(spawnedAt, end);
        }
    }

    public record PhaseTransition(
            Phase from, Phase to, Instant timestamp, String reason
    ) {}

    public enum TerminationReason {
        SUCCESS, MAX_TURNS_REACHED, BUDGET_EXHAUSTED,
        STOP_CONDITION, ERROR_ESCALATION, TIMEOUT
    }
}
```

> **Hinweis:** Der Agent Lifecycle ist das Fundament für das Execution Budget (Kapitel 7): Jede Phase verbraucht Tokens und Zeit. Context Build und Planning verbrauchen typischerweise 20–30% des Token-Budgets, bevor die eigentliche Execution beginnt.

7. Execution Model

> **Hinweis:** Abgrenzung: Kapitel 6 (Lifecycle) beschreibt die Phasen eines einzelnen Agenten. Dieses Kapitel beschreibt, wie der Orchestrator mehrere Agenten als Workflow plant, steuert und terminiert. Kapitel 15 (Workflows) zeigt die konkreten Aufruf-Patterns.

Das Execution Model beschreibt, wie der Orchestrator Aufgaben plant, verteilt, überwacht und terminiert. Es ist das operative Herzstück des Agentensystems und setzt die architektonischen Prinzipien AP-2 (Deterministic Execution) und AP-3 (Governance by Design) in die Praxis um.

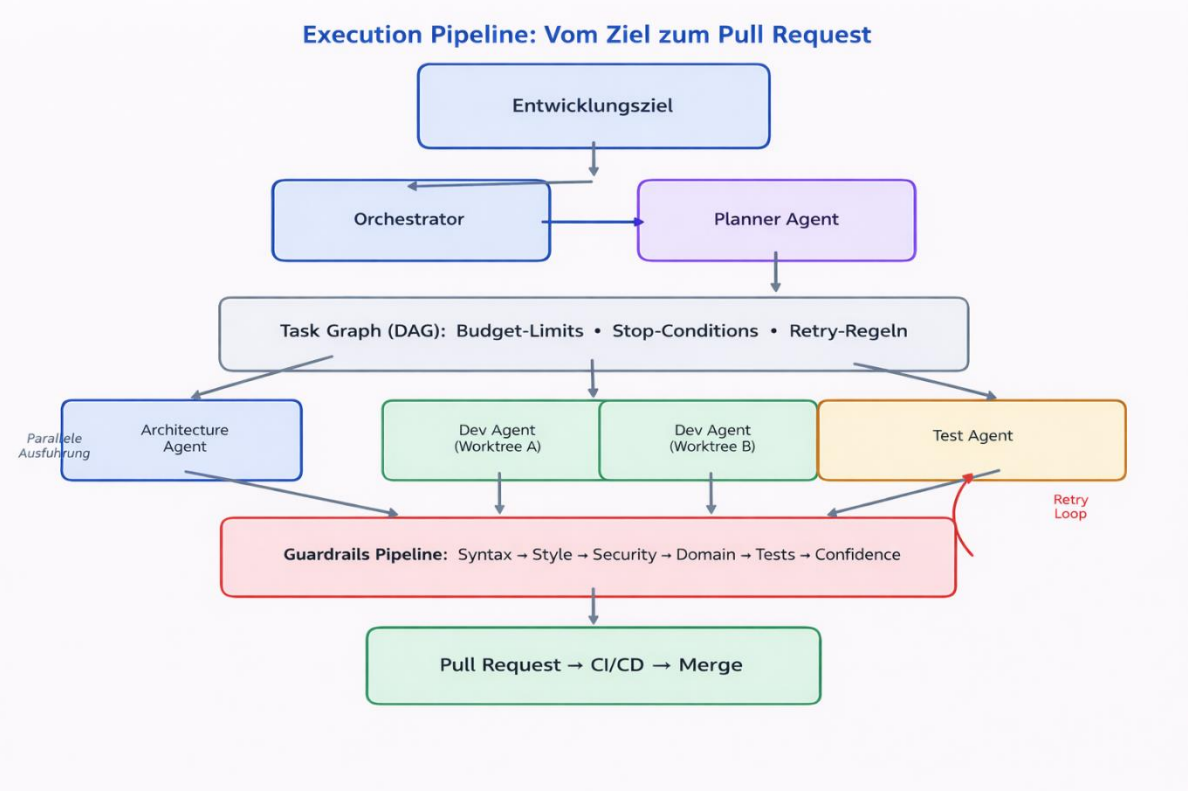


Abbildung 6: Execution Pipeline – Vom Entwicklungsziel über den Task Graph bis zum Pull Request

7.1 Task Graph

Jeder Workflow wird intern als gerichteter azyklischer Graph (DAG) repräsentiert. Knoten sind Tasks, Kanten sind Abhängigkeiten. Der Orchestrator traversiert den Graphen und startet Tasks, sobald alle Vorgänger abgeschlossen sind:

```
// === Task Graph Modell ===
public record TaskGraph(
        String workflowId,
        Map<String, TaskNode> nodes,
        Map<String, Set<String>> edges   // taskId -> dependsOn
) {
    public record TaskNode(
            String taskId,
            String agentType,
            String model,
            String prompt,
            TaskStatus status,
            @Nullable String resultRef
    ) {}

    public enum TaskStatus { PENDING, READY, RUNNING, COMPLETED, FAILED, SKIPPED }

    // Nächste ausführbare Tasks (alle Abhängigkeiten erfüllt)
    public List<TaskNode> readyTasks() {
```

```
        return nodes.values().stream()
                .filter(n -> n.status() == TaskStatus.PENDING)
                .filter(n -> {
                    Set<String> deps = edges.getOrDefault(n.taskId(), Set.of());
                    return deps.stream().allMatch(depId ->
                            nodes.get(depId).status() == TaskStatus.COMPLETED);
                }).toList();
    }

    public boolean isTerminal() {
        return nodes.values().stream().allMatch(n ->
                n.status() == TaskStatus.COMPLETED
                || n.status() == TaskStatus.FAILED
                || n.status() == TaskStatus.SKIPPED);
    }
}
```

7.2 Runtime Execution Flow

Nachdem der Task Graph erzeugt wurde, stellt sich die Frage, wie ein konkreter Entwicklungsauftrag während der Laufzeit durch das Agentensystem verarbeitet wird.

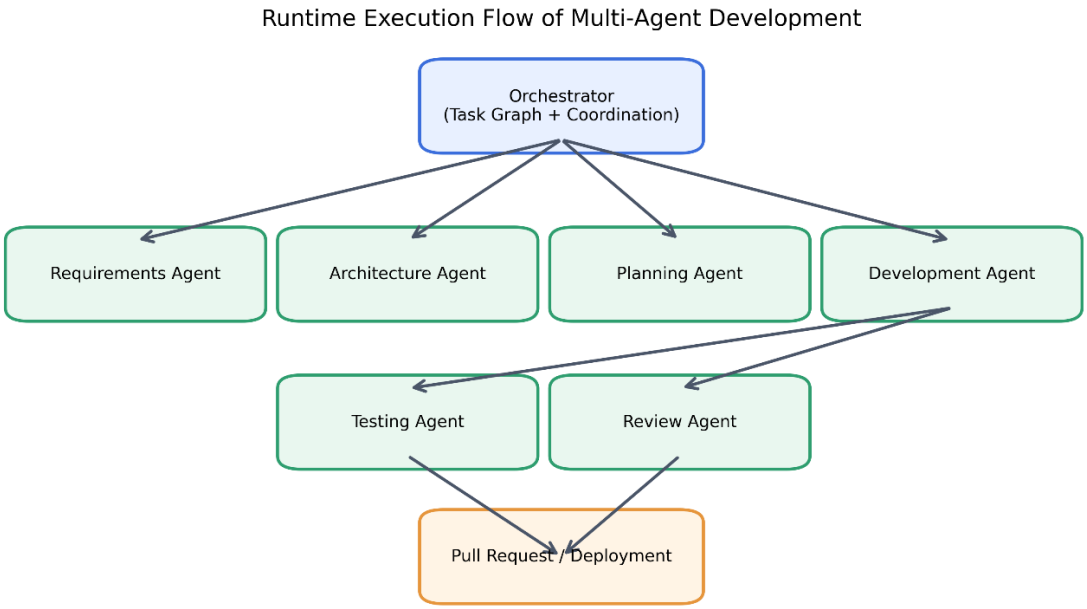


Abbildung 7: Laufzeitablauf eines Entwicklungsauftrags

Die Abbildung zeigt den typischen Ablauf eines Entwicklungsauftrags innerhalb eines agentischen Entwicklungssystems.

Ein Entwicklungsziel wird zunächst vom Orchestrator entgegengenommen, der als zentrale Steuerungseinheit fungiert. Der Orchestrator erzeugt einen Task Graph, der die notwendigen Arbeitsschritte und deren Abhängigkeiten beschreibt.

Anschließend delegiert der Orchestrator Teilaufgaben an spezialisierte Agenten. Requirements-, Architektur- und Planungsagenten analysieren zunächst Anforderungen und Systemkontext. Darauf aufbauend implementiert der Development-Agent die erforderlichen Änderungen im Code.

Die erzeugten Artefakte werden anschließend durch Testing- und Review-Agenten validiert. Diese prüfen Funktionalität, Codequalität und Sicherheitsanforderungen. Erst nach erfolgreicher Validierung wird ein Pull Request erzeugt oder ein Deployment vorbereitet. Dieser Ablauf ermöglicht eine klare Trennung der Verantwortlichkeiten zwischen den Agenten und erlaubt sowohl sequenzielle als auch parallele Ausführung einzelner Schritte.

7.3 State Machine & Stop-Conditions

Der Orchestrator implementiert eine State Machine, die den Workflow-Zustand verwaltet. Zustandswechsel werden durch Agenten-Ergebnisse, Budgetgrenzen oder externe Signale ausgelöst. Stop-Conditions definieren, wann ein Workflow vorzeitig beendet wird:

Stop-Condition	Auslöser	Verhalten
Budget exhausted	Token- oder Kosten-Budget überschritten	Alle laufenden Agenten beenden, Ergebnisse sichern
Critical failure	Agent meldet CRITICAL Finding	Workflow stoppen, Human-in-the-Loop eskalieren
Max retries exceeded	Agent scheitert N-mal am gleichen Task	Task als FAILED markieren, Workflow fortsetzen
Timeout	Wanduhrzeit überschritten	Laufende Agenten terminieren, Teilresultate sichern
External signal	Menschliche Intervention / CI-Abbruch	Graceful Shutdown aller Agenten
Quality gate failed	Code Coverage < Schwellenwert	Zurück zur Testing-Phase, Retry-Counter erhöhen

7.3 Execution Budget

Das Execution Budget ist ein zweilagiges Sicherheitsnetz: Es begrenzt sowohl einzelne Tasks (max_turns, timeout) als auch den Gesamtworkflow (max_cost, max_duration). Budgets werden pro Sprint, Feature oder Team definiert:

```
// === Execution Budget ===
public record ExecutionBudget(
        int maxTurnsPerTask,
        Duration maxDurationPerTask,
        BigDecimal maxCostPerTask,
        BigDecimal maxCostPerWorkflow,
        int maxRetriesPerTask,
        int maxParallelAgents
) {
    public static ExecutionBudget standard() {
        return new ExecutionBudget(15, Duration.ofMinutes(5),
                new BigDecimal("5.00"), new BigDecimal("50.00"), 3, 4);
    }

    public static ExecutionBudget conservative() {
        return new ExecutionBudget(10, Duration.ofMinutes(3),
                new BigDecimal("2.00"), new BigDecimal("20.00"), 2, 2);
    }

    public static ExecutionBudget aggressive() {
        return new ExecutionBudget(25, Duration.ofMinutes(10),
                new BigDecimal("10.00"), new BigDecimal("100.00"), 5, 7);
    }
}
```

7.4 Retry-Strategie

Nicht jeder Fehlschlag erfordert menschliche Intervention. Die Retry-Strategie definiert, welche Fehler automatisch wiederholt werden, mit welcher Eskalation und bis zu welchem Limit:

Fehlerart	Retry?	Strategie	Eskalation
Kompilierungsfehler	Ja (max 3x)	Agent korrigiert selbst	Nach 3x: Review-Agent
Test-Failure	Ja (max 5x)	Iterative Korrektur	Nach 5x: Human-in-the-Loop
Security Finding (CRITICAL)	Nein	Sofortige Eskalation	Human-in-the-Loop + Audit
Domain-Verletzung	Ja (max 2x)	Glossar neu laden	Nach 2x: Architecture-Agent
API/Tool Timeout	Ja (exponential)	1s, 2s, 4s Backoff	Nach 3x: Task als FAILED
Halluzinierte API	Nein	Sofortige Korrektur	Review-Agent mit opus

8. Memory Architecture

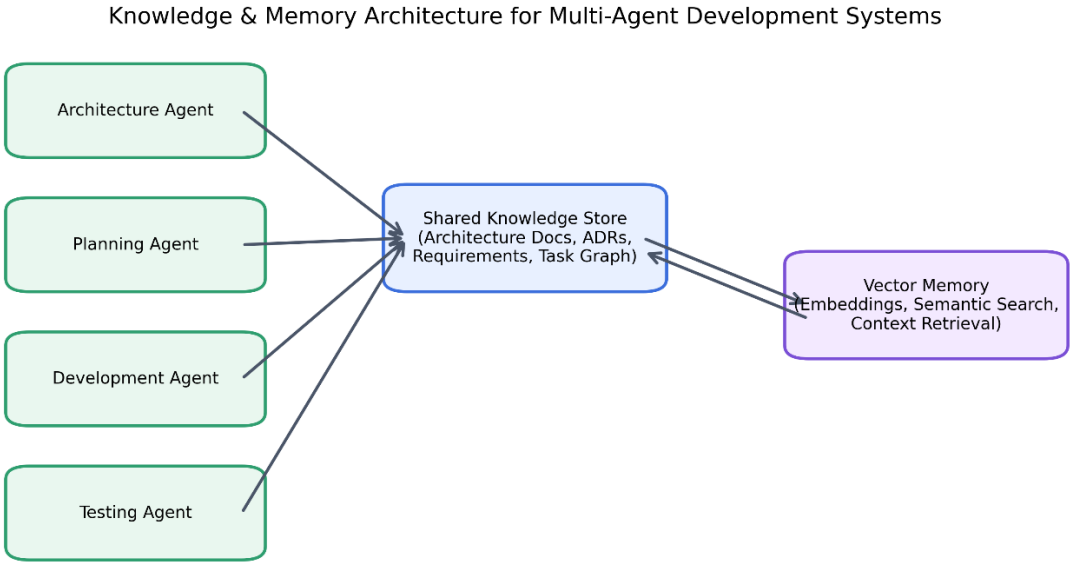


Abbildung 8: Wissens- und Speicherarchitektur eines Multi-Agent-Entwicklungssystems

Agentische Entwicklungssysteme benötigen eine gemeinsame Wissensbasis, damit spezialisierte Agenten Informationen austauschen können, ohne ihre Kontextisolation aufzugeben.

Der Shared Knowledge Store fungiert als zentrale Wissensquelle für Architekturentscheidungen, Anforderungen, Implementierungspläne und weitere Artefakte. Agenten lesen daraus Kontextinformationen und schreiben ihre Ergebnisse zurück, sodass nachfolgende Agenten darauf aufbauen können.

Ergänzend kann eine Vektor-basierte Memory-Komponente eingesetzt werden, um semantische Suche und kontextuelle Retrieval-Mechanismen zu ermöglichen. Dadurch können Agenten relevante Dokumente, Architekturentscheidungen oder Codefragmente auf Basis semantischer Ähnlichkeit finden und in ihre Entscheidungsprozesse einbeziehen.

> **Hinweis:** Ohne geteilten Zustand läuft die Kommunikation nur über den Orchestrator. Das ist sauber, aber begrenzt. Die Memory Architecture löst dieses Problem mit einem mehrschichtigen Modell.

Die Memory Architecture des Agentensystems definiert, wie Wissen gespeichert, geteilt und über den Lebenszyklus eines Workflows hinweg erhalten bleibt. Das Modell unterscheidet fünf Speicherschichten mit unterschiedlicher Lebensdauer, Sichtbarkeit und Zugriffsgeschwindigkeit:

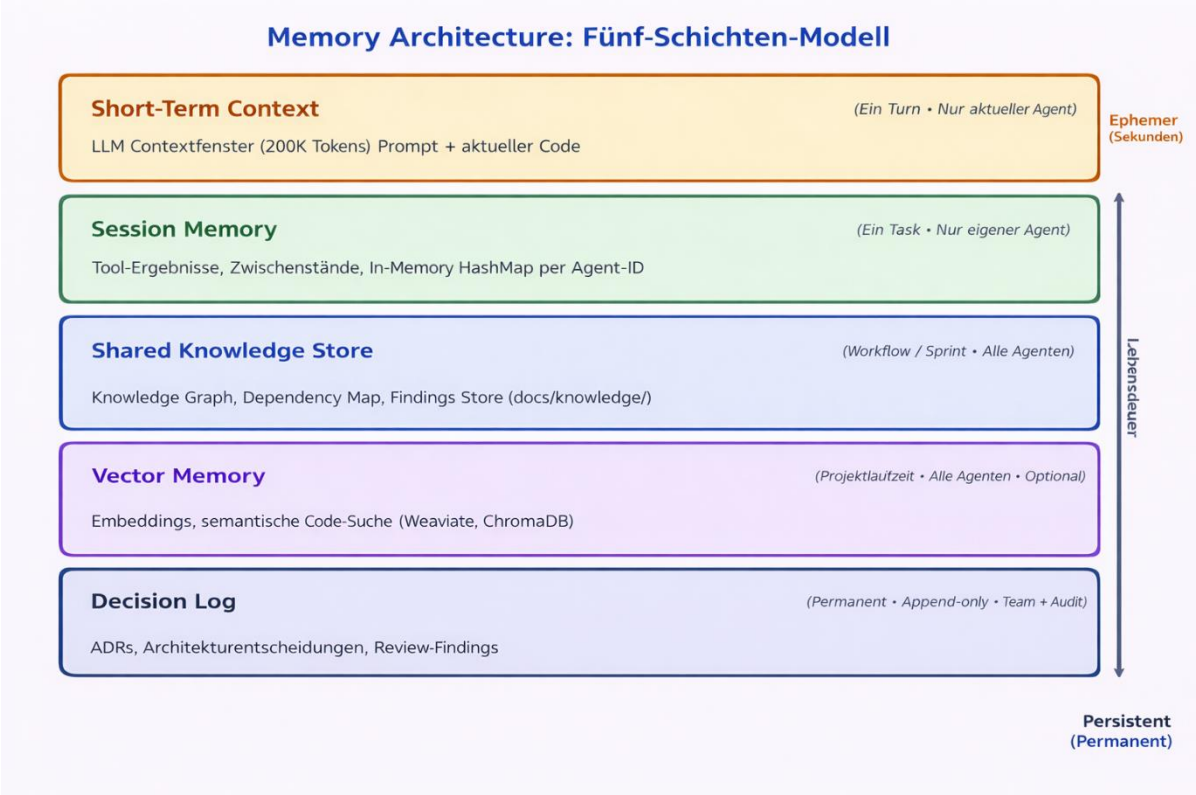


Abbildung 9: Memory Architecture – Fünf-Schichten-Modell von ephemer bis persistent

Schicht	Lebensdauer	Sichtbarkeit	Implementierung
Short-Term Context	Ein Turn	Nur der aktuelle Agent	LLM Context Window (200K Tokens)
Session Memory	Ein Task	Agent während gesamter Sitzung	In-Memory HashMap per Agent-ID
Shared Knowledge Store	Workflow / Sprint	Alle Agenten des Workflows	JSON in docs/knowledge/ (dateisystembasiert)
Vector Memory	Projektlaufzeit	Alle Agenten	Embeddings in Vector-DB (optional, z.B. Weaviate)
Decision Log	Permanent	Team + Audit	Append-only Markdown in docs/decisions/

8.1 Short-Term Context & Session Memory

Der Short-Term Context ist das LLM Context Window – maximal 200K Tokens bei Claude. Alles, was der Agent in einem Turn "sieht", existiert nur hier. Session Memory erweitert dies über mehrere Turns innerhalb eines Tasks: Der Agent kann sich an seine eigenen früheren Tool-Aufrufe und deren Ergebnisse erinnern, aber nicht an Informationen anderer Agenten.

8.2 Shared Knowledge Store

Der Shared Knowledge Store ist die zentrale Wissensbasis für die agentenübergreifende Zusammenarbeit. Er besteht aus fünf Komponenten:

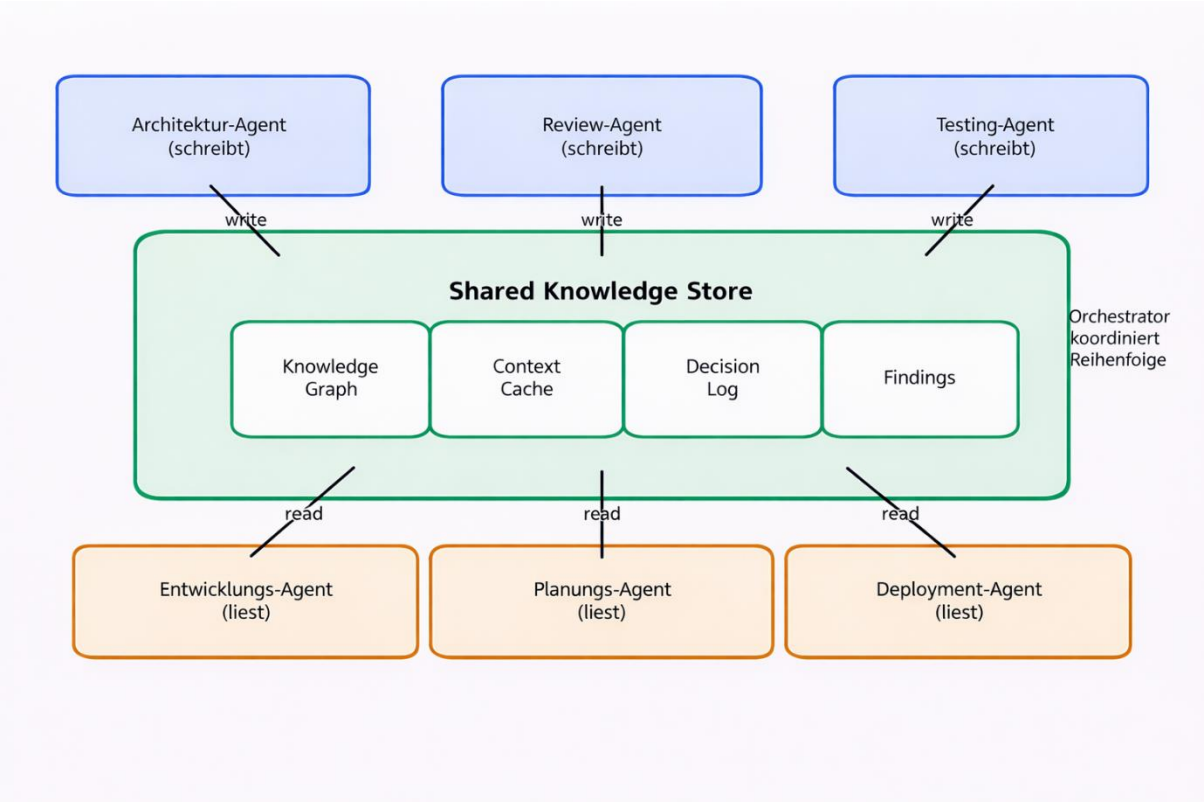


Abbildung 10: Shared Knowledge Store – Zentrale Wissensbasis für Agenten

Komponente	Beschreibung	Zugriffsmuster
Knowledge Graph	Entitäten und Beziehungen im Code	Write: architecture-agent | Read: alle
Context Cache	Kurzfristiger Sitzungs-Kontext	Write/Read: aktiver Agent
Decision Log	Architekturentscheidungen (ADRs)	Write: architecture-agent | Read: alle | Append-only
Dependency Map	Service-Abhängigkeiten	Write: architecture-agent | Read: dev-agent, deploy-agent
Findings Store	Review- und Test-Findings	Write: review-agent, test-agent | Read: dev-agent, orchestrator

Java-Beispiel: Shared Knowledge Store Client

```java
// === Shared Knowledge Store Client ===
@Component @Slf4j
public class KnowledgeStoreClient {

    private final ObjectMapper objectMapper;
    private final Path knowledgeBase;

    public KnowledgeStoreClient(
            @Value("${knowledge.base-path:docs/knowledge}") String basePath) {
        this.objectMapper = new ObjectMapper().registerModule(new JavaTimeModule());
        this.knowledgeBase = Path.of(basePath);
    }

    public <T> void store(String domain, String key, T value) {
        try {
```

```
            Path filePath = knowledgeBase.resolve(domain).resolve(key + ".json");
            Files.createDirectories(filePath.getParent());
            KnowledgeEntry<T> entry = new KnowledgeEntry<>(
                    key, value, Instant.now(), Thread.currentThread().getName());
            objectMapper.writerWithDefaultPrettyPrinter()
                    .writeValue(filePath.toFile(), entry);
            log.info("Knowledge stored: {}/{}", domain, key);
        } catch (IOException e) {
            throw new KnowledgeStoreException("Failed to store: " + domain + "/" + key, e);
        }
    }

    public <T> Optional<KnowledgeEntry<T>> retrieve(
            String domain, String key, Class<T> type) {
        Path filePath = knowledgeBase.resolve(domain).resolve(key + ".json");
        if (!Files.exists(filePath)) return Optional.empty();
        try {
            JavaType entryType = objectMapper.getTypeFactory()
                    .constructParametricType(KnowledgeEntry.class, type);
            return Optional.of(objectMapper.readValue(filePath.toFile(), entryType));
        } catch (IOException e) {
            log.warn("Failed to read: {}/{}", domain, key, e);
            return Optional.empty();
        }
    }

    public record KnowledgeEntry<T>(String key, T value, Instant storedAt, String storedBy) {}
}
```

8.3 Vector Memory (optional)

Für große Codebasen mit Millionen Zeilen Code ist ein dateibasierter Shared Knowledge Store zu langsam für semantische Suchen. Vector Memory ergänzt den Store um eine Embedding-basierte Suche, die ähnliche Code-Patterns, ähnliche Architekturentscheidungen oder relevante Findings finden kann, ohne den exakten Schlüssel zu kennen.

Die Implementierung nutzt eine lokale Vector-Datenbank (z.B. Weaviate, ChromaDB) mit einem deutschen Embedding-Modell (z.B. deepset-mxbai-embed-de-large-v1). Dies ist insbesondere für Behörden-Projekte relevant, bei denen Cloud-basierte Embedding-Services aus Datenschutzgründen nicht in Frage kommen.

9. Failure Handling & Resilience

> **Hinweis:** In Enterprise-Umgebungen ist nicht die Frage ob, sondern wann ein Agent scheitert. Ein robustes Failure-Handling-Modell ist der Unterschied zwischen einem Prototyp und einem produktionsreifen System.

Das Failure-Handling-Modell definiert fünf Eskalationsstufen, die je nach Schwere und Art des Fehlers greifen. Das Modell folgt dem Prinzip "automatisiere was möglich, eskaliere was nötig":

#	Strategie	Beschreibung	Anwendungsfall
1	Retry	Automatische Wiederholung mit gleicher oder angepasster Konfiguration. Exponential Backoff für transiente Fehler.	Kompilierungsfehler, API-Timeouts, flaky Tests
2	Rollback	Automatisches Zurücksetzen auf den letzten konsistenten Zustand über Git Worktree Reset.	Fehlgeschlagene Refactorings, kaputte Abhängigkeiten
3	Escalation	Weiterleitung an einen spezialisierten Agent (z.B. review-agent mit opus für Halluzinations-Prüfung).	LOW Confidence Score, wiederkehrende Fehler
4	Human Intervention	Eskalation an einen menschlichen Entwickler mit vollständigem Kontext (Findings, Logs, Diff).	Security CRITICAL, Domain-Verletzungen, Budget-Erschöpfung
5	Compensation	Gegenläufige Aktion zur Wiederherstellung eines konsistenten Gesamtzustands bei verteilten Workflows.	Wenn parallele Worktrees inkonsistent werden

Java-Beispiel: Failure Handler mit Eskalationslogik

```
// === Failure Handler ===
@Component @Slf4j
public class AgentFailureHandler {

    private final KnowledgeStoreClient knowledgeStore;

    public sealed interface FailureAction {
        record Retry(int attempt, int maxAttempts, Duration delay) implements FailureAction {}
        record Rollback(String worktreeId, String commitRef) implements FailureAction {}
        record Escalate(String targetAgent, String model, String context) implements FailureAction
{}
        record HumanIntervention(String summary, List<String> findings,
                                 String diffRef) implements FailureAction {}
        record Compensate(List<String> compensationTasks) implements FailureAction {}
    }

    public FailureAction determineAction(AgentFailure failure) {
        return switch (failure.severity()) {
            case TRANSIENT -> {
                if (failure.retryCount() < 3)
                    yield new FailureAction.Retry(failure.retryCount() + 1, 3,
                            Duration.ofSeconds((long) Math.pow(2, failure.retryCount())));
                yield new FailureAction.Escalate("review-agent", "opus",
                        "Transient failure after 3 retries: " + failure.message());
            }
            case RECOVERABLE -> {
                if (failure.type() == FailureType.COMPILATION_ERROR && failure.retryCount() < 3)
                    yield new FailureAction.Retry(failure.retryCount() + 1, 3,
Duration.ofSeconds(1));
                yield new FailureAction.Rollback(failure.worktreeId(), failure.lastGoodCommit());
            }
            case CRITICAL -> new FailureAction.HumanIntervention(
                    failure.message(), failure.findings(), failure.diffRef());
            case FATAL -> {
                log.error("[FATAL] Agent {} failed irrecoverably: {}", failure.agentId(),
failure.message());
```

```
                yield new FailureAction.Compensate(
                        List.of("revert-worktree:" + failure.worktreeId(),
                                "notify-team:" + failure.agentId(),
                                "update-findings:" + failure.taskId()));
            }
        };
    }

    public record AgentFailure(
            String agentId, String taskId, String worktreeId,
            String lastGoodCommit, String message, String diffRef,
            FailureType type, FailureSeverity severity,
            int retryCount, List<String> findings
    ) {}

    public enum FailureType { COMPILATION_ERROR, TEST_FAILURE, SECURITY_FINDING,
            DOMAIN_VIOLATION, HALLUCINATION, TIMEOUT, UNKNOWN }
    public enum FailureSeverity { TRANSIENT, RECOVERABLE, CRITICAL, FATAL }
}
```

TEIL III: AGENTEN IM ENTWICKLUNGSLEBENSZYKLUS

10. Die sieben Lebenszyklus-Agenten

Für jede Phase des Software Development Lifecycle steht ein spezialisierter Agent bereit. Im Folgenden werden alle sieben Agenten mit ihren Task-Aufrufen und Java-Enterprise-Codebeispielen vorgestellt.

10.1 Architektur-Agent

Der Architektur-Agent analysiert die bestehende Systemarchitektur, bewertet Design-Patterns, identifiziert Anti-Patterns und erstellt ADRs. Er hat ausschließlich lesenden Zugriff auf die Codebasis (AP-1, AP-4).

```
Task(subagent_type="architecture-agent", model="opus",
    description="Spring Boot Architektur analysieren",
    prompt=""Analysiere die Spring Boot Microservice-Architektur:
    - Controller -> Service -> Repository Trennung
    - Spring Security FilterChain-Konfiguration
    - JPA Entity-Beziehungen und Fetch-Strategien
    - Exception Handling (@ControllerAdvice)
    - Anti-Patterns: N+1 Queries, Zirkuläre Beans
    Output: Architektur-Diagramm + Findings als ADR"")
```

Erzeugter Code: Schichtentrennung

```
// === Controller Layer (REST API) ===
@RestController @RequestMapping("/api/v1/payments")
@RequiredArgsConstructor @Validated
public class PaymentController {
    private final PaymentService paymentService;

    @PostMapping @ResponseStatus(HttpStatus.CREATED)
    public ResponseEntity<PaymentResponse> initiatePayment(
            @Valid @RequestBody PaymentRequest request) {
        PaymentResponse response = paymentService.initiatePayment(request);
        URI location = ServletUriComponentsBuilder.fromCurrentRequest()
                .path("/{id}").buildAndExpand(response.paymentId()).toUri();
        return ResponseEntity.created(location).body(response);
    }
}

// === Service Layer ===
@Service @RequiredArgsConstructor @Slf4j
public class PaymentService {
    private final PaymentRepository paymentRepository;
    private final PaymentValidator paymentValidator;
    private final ApplicationEventPublisher eventPublisher;

    @Transactional
    public PaymentResponse initiatePayment(PaymentRequest request) {
        paymentValidator.validate(request);
        Payment payment = Payment.create(request.amount(), request.currency(),
request.recipientIban());
        Payment saved = paymentRepository.save(payment);
        eventPublisher.publishEvent(new PaymentInitiatedEvent(saved.getId(), saved.getAmount()));
        return PaymentResponse.from(saved);
    }
}

// === Repository Layer ===
@Repository
public interface PaymentRepository extends JpaRepository<Payment, UUID> {
    @Query("SELECT p FROM Payment p JOIN FETCH p.auditEntries WHERE p.id = :id")
    Optional<Payment> findByIdWithAudit(@Param("id") UUID id);
}
```

Globales Exception Handling (RFC 7807)

```
@RestControllerAdvice @Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(PaymentNotFoundException.class)
    public ProblemDetail handleNotFound(PaymentNotFoundException ex) {
        ProblemDetail problem = ProblemDetail.forStatusAndDetail(HttpStatus.NOT_FOUND,
ex.getMessage());
        problem.setTitle("Zahlung nicht gefunden");
        problem.setType(URI.create("https://api.company.com/errors/not-found"));
        problem.setProperty("paymentId", ex.getPaymentId());
        return problem;
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ProblemDetail handleValidation(MethodArgumentNotValidException ex) {
        ProblemDetail problem = ProblemDetail.forStatusAndDetail(HttpStatus.BAD_REQUEST,
"Validierungsfehler");
        Map<String, String> errors = ex.getBindingResult().getFieldErrors().stream()
                .collect(Collectors.toMap(FieldError::getField,
                        fe -> fe.getDefaultMessage() != null ? fe.getDefaultMessage() : "ungültig",
(a,b)->a));
        problem.setProperty("fieldErrors", errors);
        return problem;
    }
}
```

10.2 Planungs-Agent

Der Planungs-Agent erstellt detaillierte Implementierungspläne mit Meilensteinen, Abhängigkeiten und Zeitschätzungen. Er erzeugt typisierte Datenstrukturen für programmatische Fortschrittsverfolgung:

```
Task(subagent_type="planning-agent", description="JWT-Auth planen",
    prompt=""Erstelle Implementierungsplan für JWT-Authentication:
    - Token-Generierung/-Validierung mit JJWT, Refresh Token Rotation
    - RBAC, PostgreSQL User/Role-Tabellen, Spring Security Integration
    Output: docs/implementations/jwt-auth-tracker.md"")
```

```
// === Implementierungs-Tracker ===
public record ImplementationPlan(
        String featureId, String title, List<Phase> phases, Instant createdAt) {

    public record Phase(int order, String name, Duration estimatedEffort,
                        Set<String> dependencies, List<Task> tasks, PhaseStatus status) {
        public boolean isBlocked() {
            return !dependencies.isEmpty() && status == PhaseStatus.PENDING;
        }
    }
    public record Task(String id, String description, String targetFile,
                       TaskPriority priority, TaskStatus status) {}
    public enum PhaseStatus { PENDING, IN_PROGRESS, COMPLETED, BLOCKED }
    public enum TaskStatus { TODO, IN_PROGRESS, DONE, FAILED }
    public enum TaskPriority { CRITICAL, HIGH, MEDIUM, LOW }

    public double completionPercentage() {
        long total = phases.stream().flatMap(ph -> ph.tasks().stream()).count();
        long done = phases.stream().flatMap(ph -> ph.tasks().stream())
                .filter(t -> t.status() == TaskStatus.DONE).count();
        return total == 0 ? 0 : (double) done / total * 100;
    }
}
```

10.3 Requirements-Agent

Der Requirements-Agent erstellt eine Requirements Traceability Matrix (RTM), die jede Anforderung mit Code, Tests und Akzeptanzkriterien verknüpft:

```
// === Requirements Traceability Matrix ===
public record TraceabilityMatrix(String sprintId, List<TracedRequirement> requirements) {
    public record TracedRequirement(
            String jiraKey, String title, RequirementStatus status,
            List<AcceptanceCriterion> criteria, List<CodeReference> implementations,
            List<CodeReference> tests, Set<String> gaps) {
        public boolean isFullyTraced() {
            return gaps.isEmpty() && !implementations.isEmpty() && !tests.isEmpty();
        }
    }
    public record AcceptanceCriterion(String id, String description, boolean isCovered) {}
    public record CodeReference(String filePath, String className, int lineNumber) {}

    public List<TracedRequirement> untracedRequirements() {
        return requirements.stream().filter(r -> !r.isFullyTraced()).toList();
    }
}
```

10.4 Entwicklungs-Agent

Der Entwicklungs-Agent implementiert Features nach den in CLAUDE.md definierten Standards. Das folgende Beispiel zeigt das Strategy Pattern für einen Notification-Service:

```
// === Strategy Interface (Sealed) ===
public sealed interface NotificationChannel
        permits EmailChannel, SmsChannel, PushChannel {
    CompletableFuture<NotificationResult> send(NotificationRequest request);
    NotificationType getType();
    boolean supports(NotificationRequest request);
}

// === E-Mail Channel ===
@Component @Slf4j
public final class EmailChannel implements NotificationChannel {
    private final JavaMailSender mailSender;
    private final TemplateEngine templateEngine;

    @Override @Async("notificationExecutor")
    @Retryable(retryFor = MailSendException.class, maxAttempts = 3,
        backoff = @Backoff(delay = 1000, multiplier = 2.0))
    public CompletableFuture<NotificationResult> send(NotificationRequest request) {
        log.info("Sending email to {}", request.recipient());
        MimeMessage msg = mailSender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(msg, true);
        helper.setTo(request.recipient());
        helper.setSubject(request.subject());
        helper.setText(renderTemplate(request), true);
        mailSender.send(msg);
        return CompletableFuture.completedFuture(
                NotificationResult.success(getType(), request.id()));
    }
    // ...
}

// === Notification Service (Orchestrator) ===
@Service @Slf4j
public class NotificationService {
    private final List<NotificationChannel> channels;
    private final NotificationHistoryRepository historyRepo;

    @Transactional
    public List<NotificationResult> sendNotification(NotificationRequest request) {
        List<NotificationChannel> applicable = channels.stream()
                .filter(ch -> ch.supports(request)).toList();
        if (applicable.isEmpty())
            throw new NoSuitableChannelException("Kein passender Kanal");
        List<CompletableFuture<NotificationResult>> futures =
                applicable.stream().map(ch -> ch.send(request)).toList();
        List<NotificationResult> results = futures.stream()
                .map(CompletableFuture::join).toList();
        results.forEach(r -> historyRepo.save(NotificationHistory.from(request, r)));
        return results;
    }
}
```

10.5 Testing-Agent

Der Testing-Agent nutzt JUnit 5, Testcontainers und WireMock für iterative Testsuiten:

```
// === Unit Test mit Mockito ===
@ExtendWith(MockitoExtension.class)
class NotificationServiceTest {
    @Mock private NotificationHistoryRepository historyRepo;
    @Mock private EmailChannel emailChannel;
    @InjectMocks private NotificationService service;

    @Test @DisplayName("Wählt korrekte Kanäle basierend auf Request")
    void shouldSelectCorrectChannels() {
        var request = new NotificationRequest(null, "test@example.com", "User",
                "Betreff", "Inhalt", Set.of(NotificationType.EMAIL), Map.of());
        when(emailChannel.supports(request)).thenReturn(true);
        when(emailChannel.send(request)).thenReturn(CompletableFuture.completedFuture(
                NotificationResult.success(NotificationType.EMAIL, request.id())));
        var results = service.sendNotification(request);
        assertThat(results).hasSize(1);
        assertThat(results.get(0).isSuccess()).isTrue();
    }
}

// === Integrationstest mit Testcontainers ===
@SpringBootTest @Testcontainers @ActiveProfiles("test")
class NotificationIntegrationTest {
    @Container static PostgreSQLContainer<?> postgres =
            new PostgreSQLContainer<>("postgres:16-alpine");

    @DynamicPropertySource
    static void configure(DynamicPropertyRegistry r) {
        r.add("spring.datasource.url", postgres::getJdbcUrl);
        r.add("spring.datasource.username", postgres::getUsername);
        r.add("spring.datasource.password", postgres::getPassword);
    }
}
```

10.6 Review-Agent

```
Task(subagent_type="review-agent", model="opus",
    prompt=""Review des Notification-Service:
    1. Thread-Safety in Singleton-Beans
    2. Resource Leaks, JPA N+1 Queries
    3. Security: Input Validation, SQL Injection
    4. Java 21: Records, Sealed Interfaces
    Output: [CRITICAL/WARNING/INFO] Datei:Zeile - Beschreibung"")
```

10.7 Deployment-Agent

```
// Kubernetes Deployment
apiVersion: apps/v1
kind: Deployment
metadata: { name: notification-service }
spec:
  replicas: 2
  template:
    spec:
      containers:
      - name: notification-service
        image: registry.company.com/notification-service:1.0.0
        resources:
          requests: { memory: "512Mi", cpu: "500m" }
          limits: { memory: "1Gi", cpu: "1000m" }
        readinessProbe:
          httpGet: { path: /actuator/health/readiness, port: 8080 }
        livenessProbe:
          httpGet: { path: /actuator/health/liveness, port: 8080 }
```

TEIL IV: ENTERPRISE-ERWEITERUNGEN

11. Domain-Driven Design Integration

> **Hinweis:** Ohne fachliches Domänenmodell generieren Agenten technisch korrekten, aber fachlich fragwürdigen Code. DDD-Integration ist für Enterprise-Projekte unerlässlich.

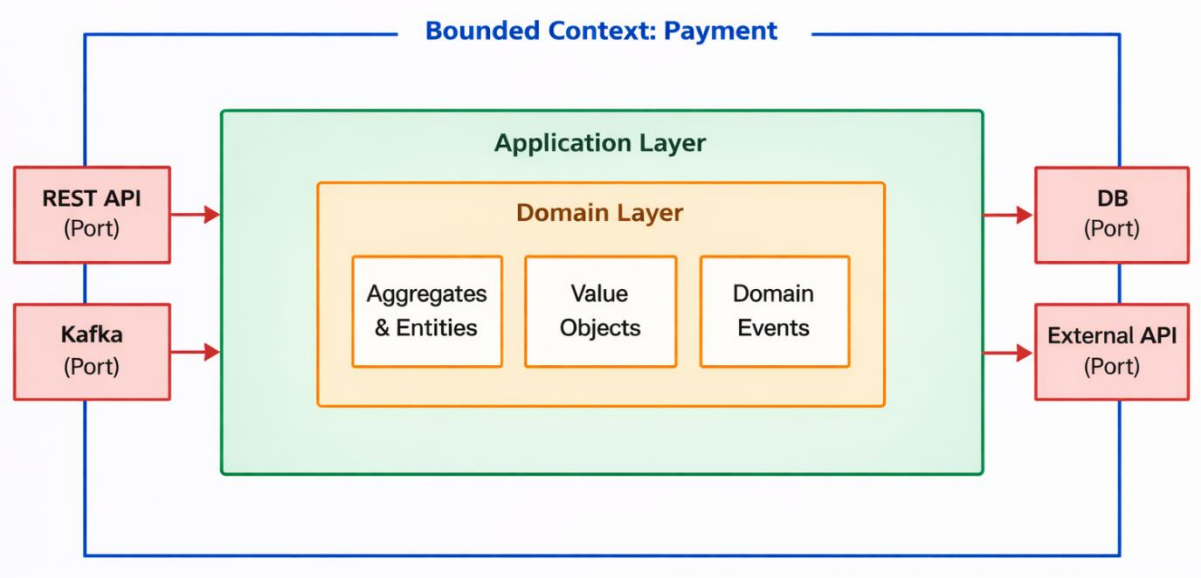


Abbildung 11: Hexagonale Architektur mit DDD und Agenten-Zuordnung

DDD-Konzept	Agenten-Integration	Beispiel
Bounded Context	Agenten respektieren strikt die Context-Grenzen	payment-context/, user-context/
Ubiquitous Language	Glossar in docs/domain/glossary.md	Zahlungsauslösung statt triggerPayment
Aggregates	Dev-Agent erstellt Invarianten und Consistency Rules	OrderAggregate mit OrderLines
Domain Events	Kafka-Producer/Consumer für Event-Driven Architecture	PaymentInitiatedEvent
Context Map	Architecture-Agent pflegt die Beziehungskarte	docs/domain/context-map.json

Java-Beispiel: DDD-Aggregate mit Java 21

```
// === Value Object als Record ===
public record Money(@Positive BigDecimal amount, @NotNull Currency currency)
        implements Comparable<Money> {
    public Money { amount = amount.setScale(2, RoundingMode.HALF_UP); }
    public static Money of(double amount, String code) {
        return new Money(BigDecimal.valueOf(amount), Currency.getInstance(code));
    }
    public Money add(Money other) { requireSameCurrency(other);
        return new Money(amount.add(other.amount), currency); }
    private void requireSameCurrency(Money o) {
        if (!currency.equals(o.currency)) throw new CurrencyMismatchException(currency, o.currency);
    }
}

// === Aggregate Root ===
@Entity @Table(name = "orders")
public class Order extends AbstractAggregateRoot<Order> {
    @EmbeddedId private OrderId id;
    @Embedded private Money totalAmount;
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
```

```
    @JoinColumn(name = "order_id")
    private List<OrderLine> lines = new ArrayList<>();
    @Enumerated(EnumType.STRING) private OrderStatus status;

    public static Order create(OrderId id) {
        Order order = new Order();
        order.id = id; order.status = OrderStatus.DRAFT;
        order.totalAmount = Money.of(0, "EUR");
        order.registerEvent(new OrderCreatedEvent(id, Instant.now()));
        return order;
    }
    public void confirm() {
        if (lines.isEmpty()) throw new EmptyOrderException(id);
        if (status != OrderStatus.DRAFT) throw new InvalidOrderTransitionException(id, status);
        status = OrderStatus.CONFIRMED;
        registerEvent(new OrderConfirmedEvent(id, totalAmount, Instant.now()));
    }
}

// === Domain Events (Sealed Interface) ===
public sealed interface OrderEvent {
    OrderId orderId(); Instant occurredAt();
    record OrderCreatedEvent(OrderId orderId, Instant occurredAt) implements OrderEvent {}
    record OrderConfirmedEvent(OrderId orderId, Money total, Instant occurredAt) implements
OrderEvent {}
}
```

Kafka Outbox Pattern

```
@Component @Slf4j
public class OutboxEventPublisher {
    private final OutboxRepository outboxRepo;
    private final KafkaTemplate<String, String> kafka;
    private final ObjectMapper mapper;

    @TransactionalEventListener(phase = BEFORE_COMMIT)
    public void handleDomainEvent(OrderEvent event) {
        outboxRepo.save(new OutboxEntry(UUID.randomUUID(),
                event.getClass().getSimpleName(), event.orderId().value().toString(),
                mapper.writeValueAsString(event), Instant.now(), OutboxStatus.PENDING));
    }

    @Scheduled(fixedDelay = 1000) @Transactional
    public void publishPendingEvents() {
        outboxRepo.findByStatusOrderByCreatedAtAsc(OutboxStatus.PENDING).forEach(entry -> {
            try { kafka.send("order-events", entry.getAggregateId(), entry.getPayload())
                    .get(5, TimeUnit.SECONDS);
                entry.markAsPublished();
            } catch (Exception e) { entry.markAsFailed(e.getMessage()); }
        });
    }
}
```

12. AI Risk Framework & Guardrails

Beim Einsatz agentischer Entwicklungssysteme besteht eine zentrale Herausforderung darin, sicherzustellen, dass generierter Code nicht nur syntaktisch korrekt ist, sondern auch sicher, architekturkonform und fachlich valide bleibt.

Dazu wird eine mehrstufige Guardrails-Pipeline eingesetzt, die jede durch Agenten erzeugte Codeänderung automatisch validiert.

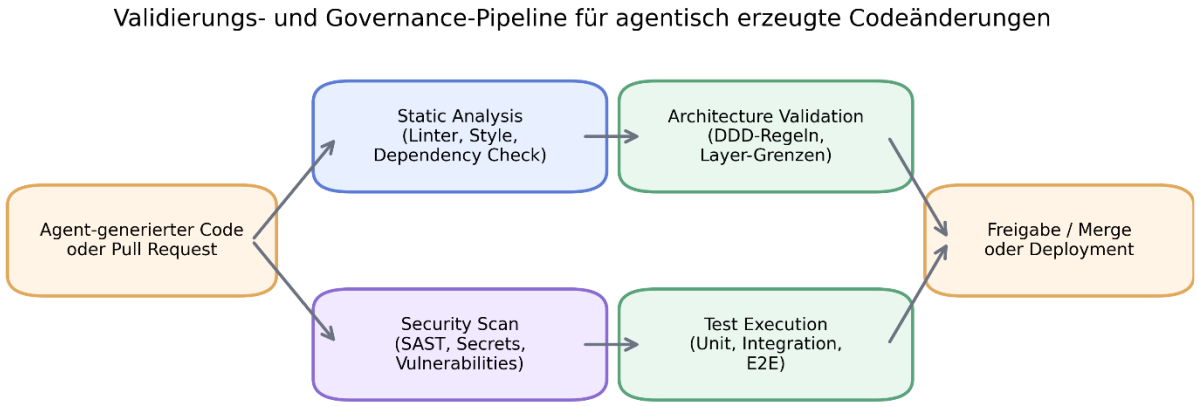


Abbildung 12: Validierungs- und Governance-Pipeline

Jede durch Agenten erzeugte Änderung durchläuft eine Reihe automatisierter Prüfungen. Dazu gehören statische Codeanalyse, Security-Scans, architektonische Validierung sowie automatisierte Tests.

Erst wenn alle Prüfungen erfolgreich abgeschlossen sind, kann eine Änderung in die Codebasis integriert oder für ein Deployment freigegeben werden.
Diese Guardrails stellen sicher, dass KI-generierter Code denselben Qualitäts- und Sicherheitsanforderungen entspricht wie manuell entwickelte Software.

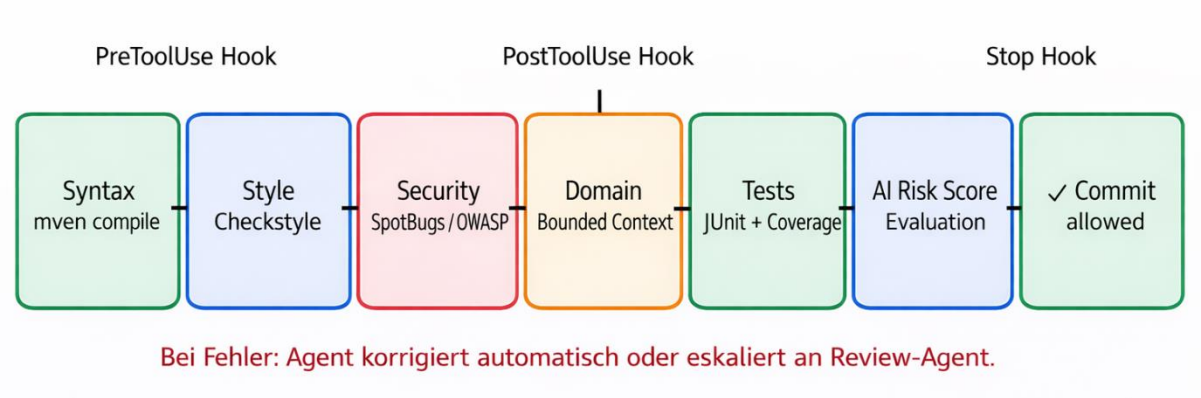


Abbildung 13: AI-Guardrails-Pipeline zur Validierung agentisch erzeugter Codeänderungen

Komponente	Funktion	Umsetzung
Halluzinations-Erkennung	Prüft ob Code auf existierenden Patterns basiert	Grep/AST-Analyse vor Commit
Schema-Validierung	Validiert JSON/YAML gegen definierte Schemas	JSON Schema im PostToolUse Hook
Security-Scan	Statische Sicherheitsanalyse	SpotBugs, OWASP Dependency Check

Domain-Prüfung	Bounded Context Einhaltung	Custom Lint gegen docs/domain/
Confidence Score	Agenten bewerten ihre eigene Sicherheit	Annotation + AOP-Aspekt
Rollback	Automatisches Zurücksetzen	Git Worktree + Cleanup

Confidence Scoring mit AOP-Eskalation

```
// === Annotation ===
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface ConfidenceScore {
    Level value();
    String rationale() default "";
    enum Level {
        HIGH(90, "Pattern existiert in Codebasis"),
        MEDIUM(65, "Best Practice, kein Vorbild"),
        LOW(30, "Unsicher, Review erforderlich");
        private final int score; private final String desc;
        Level(int s, String d) { this.score = s; this.desc = d; }
    }
}

// === AOP Aspekt ===
@Aspect @Component @Slf4j
public class ConfidenceScoreAspect {
    private final KnowledgeStoreClient knowledgeStore;

    @Around("@annotation(cs) || @within(cs)")
    public Object enforce(ProceedingJoinPoint jp, ConfidenceScore cs) throws Throwable {
        if (cs.value() == ConfidenceScore.Level.LOW) {
            log.warn("[CONFIDENCE LOW] {} - {}", jp.getSignature().toShortString(), cs.rationale());
            knowledgeStore.store("findings", "confidence-" + jp.getSignature().hashCode(),
                    new ReviewFinding(jp.getSignature().toShortString(), cs.value(),
                            cs.rationale(), Instant.now()));
        }
        return jp.proceed();
    }
}
```

Schritt	Prüfung	Bei Fehler
1. Syntax	mvn compile	Agent korrigiert automatisch
2. Style	Checkstyle, Google Java Style Guide	Agent formatiert nach
3. Security	SpotBugs + OWASP	CRITICAL-Findings markiert
4. Domain	Bounded Context, Glossar	Agent gestoppt + informiert
5. Tests	JUnit 5 + Coverage min. 80%	Agent iteriert
6. Confidence	LOW-Stellen gezählt	Weiterleitung an review-agent

13. Deployment Architektur

Während die vorherigen Kapitel die logische Architektur und das Ausführungsmodell des agentischen Entwicklungssystems beschreiben, stellt sich in der Praxis die Frage, wie ein solches System in einer Enterprise-Umgebung betrieben werden kann.

Agentische Entwicklungssysteme bestehen typischerweise aus einer zentralen Orchestrierungskomponente sowie einem Pool spezialisierter Agenten, die containerisiert betrieben und dynamisch skaliert werden können.

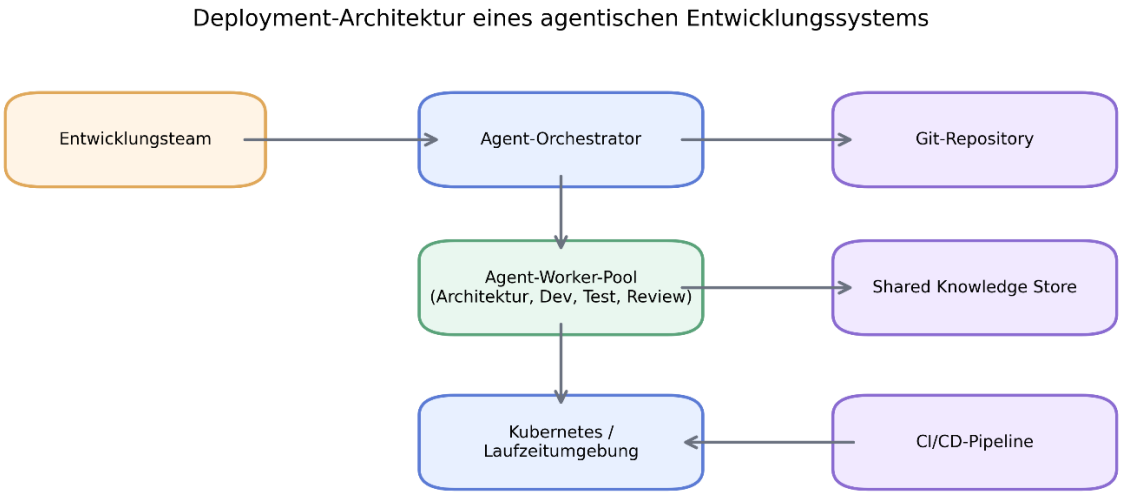


Abbildung 14: Deployment-Architektur eines agentischen Entwicklungssystems

Die Abbildung zeigt eine mögliche Infrastruktur für den Betrieb eines agentischen Entwicklungssystems in einer Enterprise-Umgebung.

Im Zentrum steht der **Agent-Orchestrator**, der Entwicklungsaufträge entgegennimmt und an einen Pool spezialisierter Agenten delegiert. Diese Agenten werden typischerweise containerisiert betrieben und können innerhalb einer **Kubernetes-Umgebung** dynamisch skaliert werden.

Die Agenten interagieren mit einem **Git-Repository**, um bestehende Codebasen zu analysieren und Änderungen vorzunehmen. Generierte Artefakte werden anschließend über eine **CI/CD-Pipeline** gebaut, getestet und validiert. Parallel greifen die Agenten auf einen **Shared Knowledge Store** zu, der Architekturentscheidungen, Anforderungen und weitere Kontextinformationen enthält.

Diese Infrastruktur ermöglicht eine **reproduzierbare, skalierbare und kontrollierte Ausführung agentischer Entwicklungsprozesse**, während bestehende Entwicklungswerkzeuge und Governance-Mechanismen weiterhin genutzt werden können.

In größeren Organisationen wird der Agent-Worker-Pool häufig über Queue- oder Workflow-Systeme gesteuert, um Priorisierung, Parallelisierung und Ressourcenmanagement zu ermöglichen.

14. Security Model

> **Hinweis:** Guardrails schützen vor fehlerhaftem Output. Das Security Model schützt das Agentensystem selbst – vor Prompt Injection, Tool Escalation, Secret Leakage und Supply-Chain-Angriffen.

Ein KI-Agentensystem mit Schreibzugriff auf die Codebasis, Bash-Zugriff und Netzwerkverbindungen ist eine erhebliche Angriffsfläche. Das Security Model adressiert fünf zentrale Bedrohungsvektoren:

Bedrohung	Risiko	Gegenmaßnahme
Prompt Injection	Manipulation des Agentenverhaltens durch eingeschleuste Anweisungen in Code-Kommentaren, Dateien oder API-Responses	Input Sanitization, System-Prompt Hardening, Ergebnis-Validierung durch separaten Review-Agent (AP-1)
Tool Escalation	Agent versucht, nicht autorisierte Tools zu nutzen oder Berechtigungsgrenzen zu überschreiten	Striktes Tool-Whitelisting per Agent in CLAUDE.md, PreToolUse-Hook blockiert nicht autorisierte Aufrufe (AP-5)
Secret Leakage	Credentials, API-Keys oder sensible Daten werden in generierten Code, Logs oder Shared Knowledge Store geschrieben	Secret-Scanner im PostToolUse-Hook (Regex + Entropy-Analyse), env-basiertes Secret Management, Audit-Trail
Supply Chain	Agent fügt kompromittierte Dependencies hinzu oder nutzt ungeprüfte Libraries	OWASP Dependency Check im Guardrail, Lockfile-Validierung, Allowlist für Dependencies in CLAUDE.md
Sandboxing	Unkontrollierter Systemzugriff durch Bash-Tool oder Datei-Operationen	Worktree-Isolation (AP-4), readonly Mounts, Netzwerk-Restriktionen, chroot für Bash-Ausführung

Berechtigungsmodell (Least Privilege)

Agent	Lesen	Schreiben	Ausführen
architecture-agent	Gesamte Codebasis	Keine	Nur-Lese Bash
planning-agent	Gesamte Codebasis	Nur Docs	Keine
dev-agent	Gesamte Codebasis	Voll	Build/Lint
test-agent	Gesamte Codebasis	Nur Tests	Test-Kommandos
review-agent	Gesamte Codebasis	Keine	Nur-Lese Bash
deploy-agent	Gesamte Codebasis	Nur IaC	Terraform/K8s

Java-Beispiel: Secret-Scanner Hook

```java
// === Secret Scanner (PostToolUse Hook) ===
public class SecretScannerHook {
    private static final List<Pattern> SECRET_PATTERNS = List.of(
        Pattern.compile("(?i)(password|secret|api[_-]?key|token)\\s*[=:]\\s*['\"][^\"]{8,}"),
        Pattern.compile("AKIA[0-9A-Z]{16}"),                  // AWS Access Key
        Pattern.compile("ghp_[0-9a-zA-Z]{36}"),               // GitHub Token
        Pattern.compile("-----BEGIN (RSA |EC )?PRIVATE KEY"), // Private Keys
        Pattern.compile("[0-9a-f]{40}")                        // High-entropy hex (SHA1-like)
    );

    public record ScanResult(boolean clean, List<Finding> findings) {
        public record Finding(String file, int line, String patternName, String masked) {}
    }

    public ScanResult scan(Path changedFile) {
        List<ScanResult.Finding> findings = new ArrayList<>();
        try {
            List<String> lines = Files.readAllLines(changedFile);
            for (int i = 0; i < lines.size(); i++) {
```

```
                for (Pattern pat : SECRET_PATTERNS) {
                    if (pat.matcher(lines.get(i)).find()) {
                        findings.add(new ScanResult.Finding(
                                changedFile.toString(), i + 1,
                                pat.pattern().substring(0, 20) + "...",
                                lines.get(i).substring(0, Math.min(40, lines.get(i).length())) +
"..."));
                    }
                }
            }
        } catch (IOException e) { throw new HookException("Scan failed", e); }
        return new ScanResult(findings.isEmpty(), findings);
    }
}
```

15. Wirtschaftlichkeit & Kostenmodell

> **Hinweis:** Agentensysteme können erhebliche API-Kosten verursachen. Ohne aktives Kostenmanagement skalieren die Ausgaben unkontrolliert. Ein transparentes Kostenmodell ist Voraussetzung für den Enterprise-Einsatz.

15.1 Token Budget Management

Modell	Input (pro 1M Tokens)	Output (pro 1M Tokens)	Typischer Einsatz
opus	$15.00	$75.00	Architektur, Security Review
sonnet	$3.00	$15.00	Entwicklung, Testing, Planung
haiku	$0.25	$1.25	Formatierung, einfache Tasks

Ein typischer Entwicklungs-Agent verbraucht 10.000–100.000 Tokens pro Task. Ein End-to-End-Workflow mit 7 Agenten kann 500.000–2.000.000 Tokens verbrauchen.

```
// === Token Budget Tracker ===
public record TokenBudget(
        String budgetId, String scope,
        BigDecimal maxCostUsd, Map<String, AgentUsage> usageByAgent) {

    public record AgentUsage(String agentType, String model,
            long inputTokens, long outputTokens, BigDecimal costUsd) {}

    public BigDecimal totalCost() {
        return usageByAgent.values().stream()
                .map(AgentUsage::costUsd).reduce(BigDecimal.ZERO, BigDecimal::add);
    }
    public double utilizationPercent() {
        return totalCost().divide(maxCostUsd, 4, RoundingMode.HALF_UP)
                .multiply(BigDecimal.valueOf(100)).doubleValue();
    }
    public boolean isExhausted() { return totalCost().compareTo(maxCostUsd) >= 0; }
    public boolean isWarning() { return utilizationPercent() >= 80.0; }
}
```

15.2 Execution Budget & Stop-Conditions

Parameter	Steuerung	Empfehlung
max_turns	Maximale API-Roundtrips pro Agent	10–25 für Entwicklung, 5–10 für Reviews
timeout	Maximale Laufzeit in Sekunden	300s Standard, 600s für komplexe Tasks
max_cost_per_task	Kostenobergrenze pro Einzeltask	$2–5 für sonnet, $10–20 für opus
max_cost_per_workflow	Kostenobergrenze Gesamtworkflow	$20–50 pro Feature
stop_on_error	Abbruch bei Kompilier-/Testfehlern	true für Deployment, false für Entwicklung

15.3 Parallelisierungskosten

Szenario	Wanduhrzeit	Kosten
Sequenziell: 7 Agenten	~45 Minuten	$8–12
Hybrid: 3 seq. + 4 parallel	~25 Minuten	$8–12 (gleich)
Maximal parallel: 7 gleichzeitig	~10 Minuten	$8–12 (gleich)
Parallel mit Retry-Schleifen	~15 Minuten	$12–20 (höher!)

Die Token-Kosten bleiben bei Parallelisierung identisch. Teurer wird es erst bei Merge-Konflikten und Retry-Schleifen. Deshalb ist Worktree-Isolation (AP-4) bei paralleler Ausführung Pflicht.

15.4 ROI-Berechnung

Kostenfaktor	Manuell (Entwicklerteam)	KI-Agenten + Review
Entwickleraufwand	40h Senior Dev × €95/h = €3.800	8h Review + Steering = €760
API-Kosten	–	~2M Tokens ≈ €14–28
Testabdeckung	Oft <60% unter Zeitdruck	>80% durch iteratives Testing
Time-to-Feature	1–2 Wochen	1–2 Tage
Gesamtkosten	€3.800+	€790–€850

Der ROI hängt stark von der Aufgabenkomplexität ab: Bei Standard-CRUD-Features ist der Hebel am größten (5–10x). Bei komplexen Architekturentscheidungen sinkt der Automatisierungsgrad, aber der Analyse-Output (ADRs, Findings) beschleunigt die menschliche Entscheidungsfindung erheblich.

Kostenoptimierungs-Strategie

```
# Budget-bewusste CLAUDE.md Konfiguration
## Cost Controls
- DEFAULT_MODEL: sonnet              # Nie opus als Default
- MAX_TURNS_DEFAULT: 15
- SPRINT_BUDGET_USD: 200
- FEATURE_BUDGET_USD: 50
- ALERT_THRESHOLD: 0.8              # Warnung bei 80%

## Modell-Eskalation (nur bei Bedarf)
- opus NUR für: ADRs, Security Reviews, Halluzinations-Prüfung
- haiku für: Formatierung, Linting, Docs ohne Fachlogik
- sonnet für: alles andere (Default)
```

TEIL V: ORCHESTRIERUNG & PRAXIS

16. Multi-Agent-Workflows

> **Hinweis:** Abgrenzung: Die theoretischen Grundlagen (Lifecycle, Execution Model, Memory) wurden in Teil II beschrieben. Dieses Kapitel zeigt die konkreten Aufruf-Patterns für sequenzielle, parallele und asynchrone Workflows.

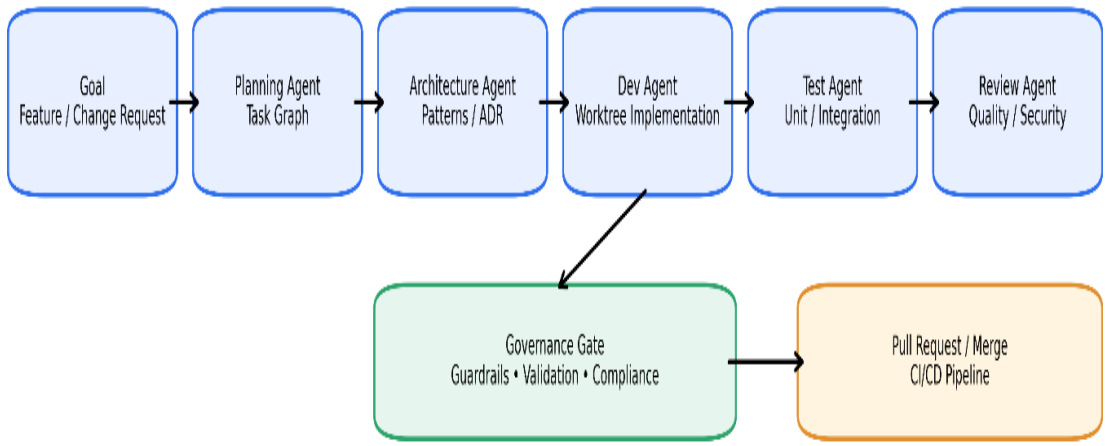


Abbildung 15: Sequenzieller Multi-Agent-Workflow im Software Development Lifecycle

Sequenzieller Workflow

```
// Phase 1: Architekturanalyse (muss zuerst laufen)
result1 = Task(subagent_type="architecture-agent",
    prompt="Analysiere Auth-Architektur, schreibe in Shared Knowledge Store...")
// Phase 2: Planung (hängt von Analyse ab)
result2 = Task(subagent_type="planning-agent",
    prompt="Lies docs/knowledge/auth-analysis.json. Erstelle Plan...")
// Phase 3: Implementierung (folgt dem Plan)
result3 = Task(subagent_type="dev-agent",
    prompt="Lies docs/implementations/auth-tracker.md. Phase 1 umsetzen...")
```

Paralleler Workflow

```
// Diese drei Agenten laufen PARALLEL:
Task(subagent_type="test-agent", prompt="JUnit 5 Tests...")
Task(subagent_type="doc-agent", prompt="OpenAPI-Dokumentation...")
Task(subagent_type="review-agent", prompt="Security-Review...")
```

Background & Wiederaufnahme

```
Task(subagent_type="test-agent", prompt="mvn test", run_in_background=True)
// Später fortsetzen:
Task(subagent_type="dev-agent", prompt="Korrigiere Fehler.", resume="agent_abc123")
```

17. Erweiterte Java Enterprise Praxisbeispiele

17.1 Spring Security mit JWT und RBAC

```
@Configuration @EnableWebSecurity @EnableMethodSecurity
@RequiredArgsConstructor
public class SecurityConfig {
    private final JwtAuthFilter jwtAuthFilter;
    private final AuthenticationProvider authProvider;

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .csrf(AbstractHttpConfigurer::disable)
            .sessionManagement(s -> s.sessionCreationPolicy(STATELESS))
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/v1/auth/**").permitAll()
                .requestMatchers("/actuator/health/**").permitAll()
                .requestMatchers("/api/v1/admin/**").hasRole("ADMIN")
                .requestMatchers("/api/v1/payments/**").hasAnyRole("USER", "ADMIN")
                .anyRequest().authenticated())
            .authenticationProvider(authProvider)
            .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)
            .build();
    }
}

// === JWT Auth Filter ===
@Component @RequiredArgsConstructor @Slf4j
public class JwtAuthFilter extends OncePerRequestFilter {
    private final JwtService jwtService;
    private final UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res,
            FilterChain chain) throws ServletException, IOException {
        String header = req.getHeader("Authorization");
        if (header == null || !header.startsWith("Bearer ")) { chain.doFilter(req, res); return; }
        String jwt = header.substring(7);
        try {
            String username = jwtService.extractUsername(jwt);
            if (username != null && SecurityContextHolder.getContext().getAuthentication() == null)
{
                UserDetails user = userDetailsService.loadUserByUsername(username);
                if (jwtService.isTokenValid(jwt, user)) {
                    var auth = new UsernamePasswordAuthenticationToken(user, null,
user.getAuthorities());
                    SecurityContextHolder.getContext().setAuthentication(auth);
                }
            }
        } catch (JwtException e) { log.warn("Invalid JWT: {}", e.getMessage()); }
        chain.doFilter(req, res);
    }
}
```

17.2 Camunda 8 mit Zeebe

```
@Component @Slf4j
public class AmountCheckWorker {
    private static final Money THRESHOLD = Money.of(10000, "EUR");

    @ZeebeWorker(type = "payment-amount-check", timeout = 30000, maxJobsActive = 10)
    public void handle(@ZeebeVariable Money amount, @ZeebeVariable String paymentId,
            JobClient client, ActivatedJob job) {
        boolean needsApproval = amount.compareTo(THRESHOLD) > 0;
        client.newCompleteCommand(job.getKey())
                .variables(Map.of("requiresApproval", needsApproval,
                        "checkResult", needsApproval ? "MANUAL_APPROVAL" : "AUTO_APPROVED"))
                .send().join();
    }
}
```

18. MCP-Server & Hooks

MCP-Server Konfiguration

```
// .claude/settings.json
{
    "mcpServers": {
        "jira-server": {
            "command": "npx", "args": ["-y", "@anthropic/mcp-jira"],
            "env": { "JIRA_URL": "https://firma.atlassian.net", "JIRA_TOKEN":
"${JIRA_API_TOKEN}" }
        },
        "postgres-server": {
            "command": "npx", "args": ["-y", "@anthropic/mcp-postgres"],
            "env": { "DATABASE_URL": "${DATABASE_URL}" }
        }
    }
}
```

Hook-Event	Auslöser	Anwendungsfall
PreToolUse	Vor jeder Werkzeugausführung	Zugriffe blockieren, Pfade validieren
PostToolUse	Nach Werkzeugabschluss	Linting, Security-Scan, Domain-Compliance
Notification	Agent benötigt Aufmerksamkeit	Slack/Teams-Alerts, Eskalation
Stop	Sitzung endet	Cleanup, Report-Generierung, Metriken

Domain-Compliance Hook

```
// === PostToolUse Hook: Bounded Context Check ===
public class DomainComplianceHook {
    private final Map<String, Set<String>> contextBoundaries;

    public record ComplianceResult(boolean compliant, List<Violation> violations) {
        public record Violation(String sourceCtx, String targetCtx, String importLine) {}
    }

    public ComplianceResult check(Path changedFile) {
        List<ComplianceResult.Violation> violations = new ArrayList<>();
        String fileCtx = resolveContext(changedFile);
        Set<String> allowed = contextBoundaries.getOrDefault(fileCtx, Set.of());

        Files.lines(changedFile).filter(l -> l.startsWith("import ")).forEach(imp -> {
            String importedCtx = resolveContextFromImport(imp);
            if (importedCtx != null && !importedCtx.equals(fileCtx) &&
!allowed.contains(importedCtx))
                violations.add(new ComplianceResult.Violation(fileCtx, importedCtx, imp));
        });
        return new ComplianceResult(violations.isEmpty(), violations);
    }
}
```

TEIL VI: ARCHITEKTURENTSCHEIDUNGEN & REFERENZ

19. Architekturentscheidungen (ADRs)

Die folgenden Architecture Decision Records dokumentieren die wichtigsten Designentscheidungen des Agentensystems. Jeder ADR folgt dem Lightweight ADR-Format nach Michael Nygard: Kontext, Entscheidung, Konsequenzen.

ADR-1: Multi-Agent vs. Single-Agent Architektur

Motivation / Kontext

KI-basierte Entwicklungssysteme lassen sich grundsätzlich in zwei Architekturen implementieren:

Single-Agent (Monolithisch): Ein einzelner LLM-Aufruf mit einem umfangreichen System-Prompt übernimmt alle Aufgaben – von der Anforderungsanalyse über die Implementierung bis zum Deployment. Der gesamte Kontext (Architekturregeln, Code-Konventionen, Domain-Wissen, Tool-Zugriffe) wird in einem einzigen Prompt-Kontext gehalten.

Multi-Agent (Spezialisiert): Mehrere Agenten mit jeweils fokussiertem Auftrag, eigenem System-Prompt, eingeschränkten Tool-Zugriffen und abgegrenztem Kontextfenster arbeiten koordiniert an einer Aufgabe. Ein Orchestrator verteilt Aufgaben und aggregiert Ergebnisse. Die Entscheidung ist architekturprägend, weil sie beeinflusst: Kontextfenster-Nutzung (und damit Antwortqualität), Token-Kosten, Parallelisierbarkeit, Fehler-Isolation, Modellwahl pro Aufgabe und Erweiterbarkeit um neue Fähigkeiten.

Aus der Praxis mit Claude Code zeigt sich: Ein monolithischer Agent mit >50.000 Token System-Prompt degradiert messbar in der Ausgabequalität, weil das Modell zwischen Architekturregeln, Test-Konventionen, Deployment-Wissen und Domain-Constraints priorisieren muss. Gleichzeitig bleiben einfache Aufgaben (Linting, Formatierung) unnötig teuer, wenn sie mit dem leistungsstärksten Modell (Opus) ausgeführt werden.

Entscheidung

Wir verwenden eine Multi-Agent-Architektur mit Hub-and-Spoke-Topologie und zentralem Orchestrator.

Agenten-Katalog

Agent	Verantwortung	Modell (Empfehlung)	Tool-Zugriffe	Kontext-Fokus
Orchestrator	Task-Dekomposition, Agent-Zuweisung, Ergebnis-Aggregation, Conflict Resolution	Opus	Alle Agenten, Shared Knowledge Store, Git	Gesamtarchitektur, Task-Graph, Agent-Status
Architecture Agent	ADR-Erstellung, Architektur-Validierung, Bounded-Context-Prüfung	Opus	Dateisystem (read), Shared Knowledge Store	ADRs, Architekturregeln, DDD-Modell
Planning Agent	Task-Dekomposition, Abhängigkeitsanalyse, Schätzung	Sonnet	Shared Knowledge Store, Issue Tracker	Anforderungen, Task-Graph, Velocity-Daten
Requirements Agent	Anforderungsanalyse, Akzeptanzkriterien, User-Story-Verfeinerung	Sonnet	Shared Knowledge Store, Dokumentation	Fachdomäne, bestehende Requirements

Agent	Verantwortung	Modell (Empfehlung)	Tool-Zugriffe	Kontext-Fokus
Development Agent	Code-Generierung, Refactoring, Implementierung	Sonnet / Opus (je nach Komplexität)	Dateisystem (read/write), Build-Tools, Terminal	Service-Code, Dependencies, API-Contracts
Testing Agent	Testgenerierung, Coverage-Analyse, Mutation Testing	Sonnet	Dateisystem, Test-Runner, Coverage-Tools	Test-Code, Fixtures, Coverage-Reports
Review Agent	Code-Review, Architektur-Compliance, Security-Check	Opus	Dateisystem (read), Guardrails-Pipeline	Diff, ADRs, Security-Policies, Style-Guides
Deployment Agent	CI/CD-Konfiguration, Infrastructure as Code, Rollout-Strategien	Sonnet / Haiku	Dateisystem, Docker, K8s-Config	Deployment-Manifeste, Helm Charts, Pipelines

Hub-and-Spoke-Topologie

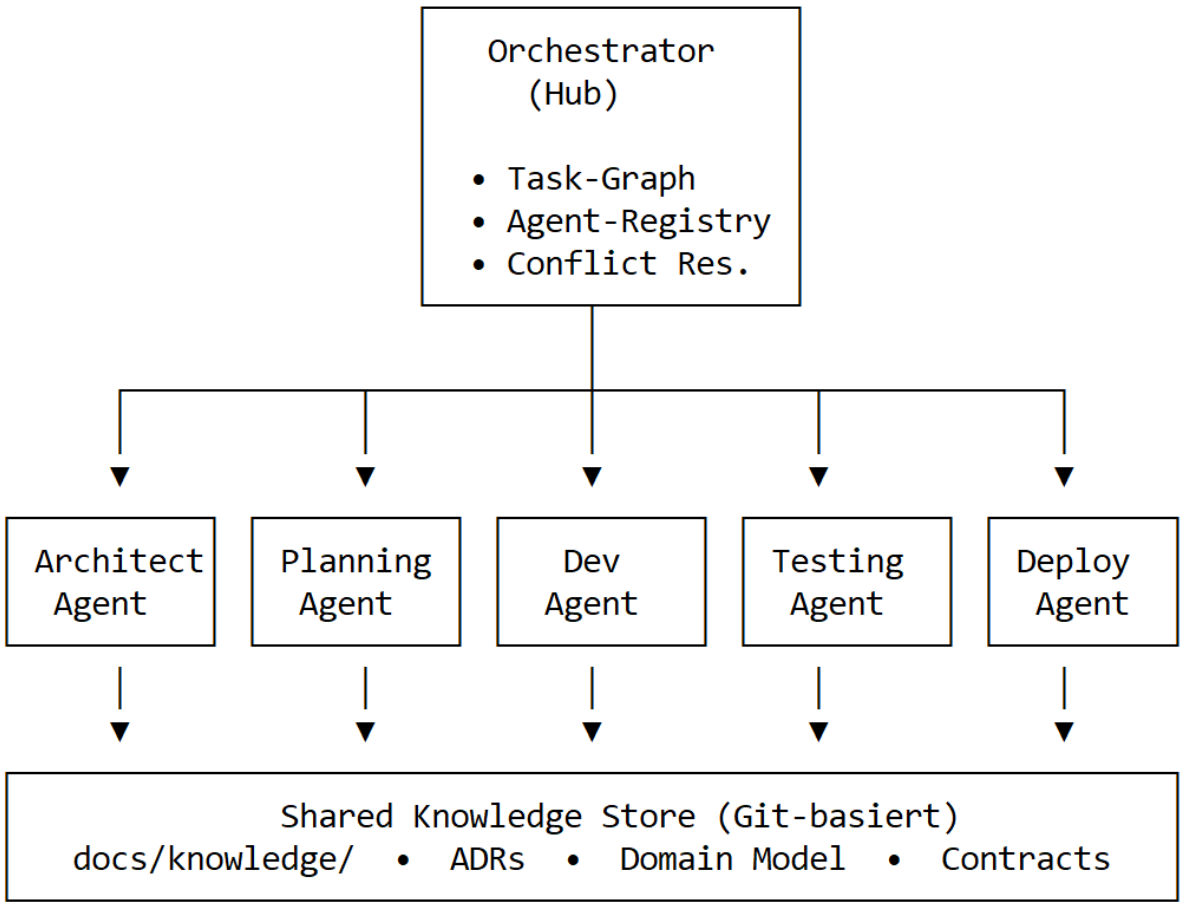


Abbildung 16: Hub and Spoke - Topologie

Kommunikationsmodell

Agenten kommunizieren **ausschließlich über den Orchestrator und den Shared Knowledge Store** – keine direkte Agent-zu-Agent-Kommunikation. Das verhindert unkoordinierte Seiteneffekte und stellt sicher, dass der Orchestrator jederzeit den Gesamtzustand kennt.

Ablauf einer typischen Feature-Implementierung:

1. Orchestrator erhält Feature-Request, ruft Requirements Agent auf
2. Requirements Agent schreibt Anforderungsdokument in Shared Knowledge Store
3. Orchestrator ruft Architecture Agent auf → prüft Architektur-Impact, erstellt/aktualisiert ADR
4. Orchestrator ruft Planning Agent auf → dekomponiert in Tasks mit Abhängigkeiten
5. Orchestrator verteilt unabhängige Tasks parallel an Development Agents (je eigener Worktree)
6. Nach Implementierung: Testing Agent generiert und führt Tests aus
7. Review Agent prüft Diff gegen ADRs, Conventions, Security-Policies
8. Bei Bestehen: Deployment Agent erstellt/aktualisiert Deployment-Konfiguration
9. Orchestrator führt Merge durch (nach bestandener Guardrails-Pipeline)

Modellwahl-Strategie

Task-Komplexität	Modell	Beispiele	Token-Kosten (relativ)
Hoch (Architekturentscheidungen, komplexe Geschäftslogik, Security-Review)	Opus	Architecture Agent, Review Agent, komplexe Dev-Tasks	1x (Referenz)
Mittel (Standard-Implementierung, Testgenerierung, Planung)	Sonnet	Development Agent, Testing Agent, Planning Agent	~0,2x
Niedrig (Formatierung, einfache Konfiguration, Boilerplate)	Haiku	Deployment Agent (simple Configs), Dokumentations-Updates	~0,04x

Diese Differenzierung senkt die Token-Kosten um 60–70 % gegenüber einer reinen Opus-Nutzung bei vergleichbarer Ergebnisqualität.

Begründung

Warum Multi-Agent statt Single-Agent?

Kontextfenster-Effizienz: Ein spezialisierter Agent benötigt nur den für seine Aufgabe relevanten Kontext. Der Architecture Agent braucht keine Test-Fixtures, der Testing Agent keine Deployment-Manifeste. Dadurch bleibt mehr Kontextfenster für die eigentliche Aufgabe – die Antwortqualität steigt messbar.

Modellwahl pro Task: Nicht jede Aufgabe rechtfertigt die Kosten des leistungsstärksten Modells. Formatierung mit Opus ist Verschwendung; eine Architekturentscheidung mit Haiku ist riskant. Multi-Agent ermöglicht die richtige Modellwahl pro Aufgabe.

Parallelisierung: Unabhängige Tasks (z. B. drei Services eines Features) können gleichzeitig von drei Development Agents bearbeitet werden. Ein Single-Agent arbeitet sequenziell.

Fehler-Isolation: Wenn der Testing Agent halluziniert, ist nur die Test-Generierung betroffen – nicht die bereits fertige Implementierung. Beim Single-Agent kann eine Halluzination den gesamten Kontext korrumpieren.

Erweiterbarkeit: Ein neuer Agent (z. B. Documentation Agent, Performance Agent) wird hinzugefügt, ohne bestehende Agenten zu ändern. Beim Single-Agent wächst der System-Prompt mit jeder neuen Fähigkeit.

Warum Hub-and-Spoke statt Mesh?

In einer Mesh-Topologie kommuniziert jeder Agent direkt mit jedem anderen. Bei n Agenten sind das n×(n-1)/2 Kommunikationskanäle. Hub-and-Spoke hat nur n Kanäle (jeder Agent zum Orchestrator). Das reduziert Komplexität, verhindert zirkuläre Abhängigkeiten und gibt dem Orchestrator die vollständige Kontrolle über die Ausführungsreihenfolge.

Warum keine Agent-zu-Agent-Kommunikation?

Direkte Kommunikation zwischen Agenten erzeugt Koordinationsprobleme: Wer hat Vorrang? Was passiert bei widersprüchlichen Anweisungen? Der Shared Knowledge Store als einzige Kommunikationsschicht stellt sicher, dass jeder Agent auf demselben, versionierten Wissensstand arbeitet.

Konsequenzen

Positive Konsequenzen

- Spezialisierung und fokussierte System-Prompts: höhere Antwortqualität pro Agent durch kleineren, relevanten Kontext
- Kostenoptimierung durch Modellwahl pro Task: 60–70 % niedrigere Token-Kosten gegenüber reiner Opus-Nutzung
- Parallelisierung unabhängiger Tasks: Reduktion der Wall-Clock-Time um Faktor 2–4 bei Multi-Service-Features
- Fehler-Isolation: Halluzination eines Agenten beeinträchtigt nicht die Ergebnisse anderer Agenten
- Erweiterbarkeit: Neue Agenten können hinzugefügt werden, ohne bestehende zu ändern (Open/Closed Principle)
- Nachvollziehbarkeit: Jeder Schritt ist einem spezifischen Agenten zuordenbar (Audit-Trail)
- Wiederverwendbarkeit: Agenten können über Projekte hinweg eingesetzt werden (z. B. derselbe Testing Agent für verschiedene Repositories)

Negative Konsequenzen

- Höhere Orchestrierungskomplexität: Der Orchestrator muss Task-Dekomposition, Abhängigkeitsanalyse und Conflict Resolution beherrschen
- Token-Overhead: Jeder Agent muss seinen Kontext neu aufbauen (System-Prompt, relevante Knowledge-Store-Einträge) – ca. 10–20 % Overhead gegenüber einem Single-Agent mit persistentem Kontext
- Debugging über Agent-Grenzen hinweg ist schwieriger: Wenn ein Feature nicht funktioniert, muss ermittelt werden, welcher Agent den Fehler verursacht hat
- Latenz: Inter-Agent-Kommunikation über Orchestrator und Shared Knowledge Store addiert Latenz (Sekunden pro Hop)
- Initialer Setup-Aufwand: System-Prompts, Tool-Konfigurationen und Guardrails für sieben Agenten statt für einen

ADR-2: Workspace Isolation via Git Worktrees

Motivation / Kontext

Die Multi-Agent-Architektur (ADR-1) ermöglicht die parallele Ausführung mehrerer Agenten – z. B. drei Development Agents, die gleichzeitig drei unabhängige Services eines Features implementieren. Diese Parallelisierung ist einer der Hauptvorteile der Architektur, setzt aber eine strikte Isolation der Arbeitsbereiche voraus.

Ohne Isolation drohen:

Race Conditions: Zwei Agenten ändern dieselbe Datei gleichzeitig → inkonsistenter Zustand, nicht kompilierbarer Code

Halbfertige Artefakte: Agent A sieht den unfertigen Zwischenstand von Agent B → generiert Code gegen instabile Interfaces

Build-Instabilität: Ein Agent startet einen Build, während ein anderer gerade Dateien schreibt → sporadische Build-Fehler

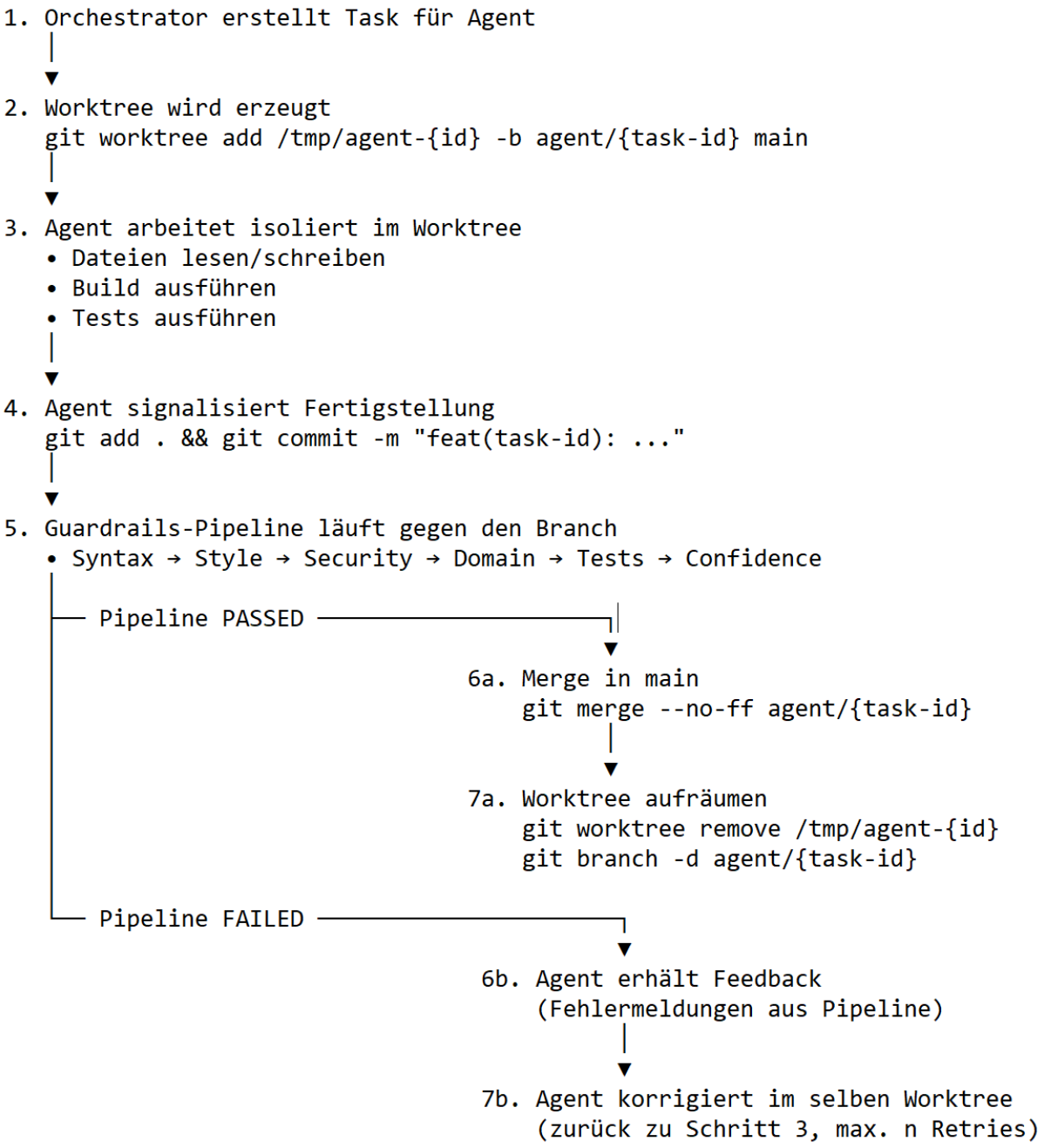
Nicht-reproduzierbare Ergebnisse: Die Reihenfolge, in der Agenten ihre Änderungen schreiben, beeinflusst das Endergebnis

Git bietet mit Worktrees einen Mechanismus, der genau dieses Problem löst: Mehrere parallele Checkouts desselben Repositories, jeder in einem eigenen Verzeichnis mit eigenem Branch, die sich eine gemeinsame .git-Datenbank teilen.

Entscheidung

Jeder parallellaufende Agent arbeitet in einem eigenen Git Worktree. Änderungen werden erst nach erfolgreicher Validierung (Guardrails-Pipeline) in den Haupt-Branch gemergt.

Worktree-Lebenszyklus



Namenskonventionen

Element	Konvention	Beispiel
Worktree-Pfad	/tmp/agent-{agent-type}-{task-id}	/tmp/agent-dev-FEAT-42-payment-service
Branch-Name	agent/{agent-type}/{task-id}	agent/dev/FEAT-42-payment-service
Commit-Message	Conventional Commits + Task-ID	feat(FEAT-42): implement PaymentService with DDD

Merge-Strategie

Szenario	Strategie
Unabhängige Dateien (verschiedene Services)	Parallele Merges – keine Konflikte zu erwarten
Überlappende Dateien (z. B. shared config, API contracts)	Sequenzieller Merge in Orchestrator-definierter Reihenfolge. Zweiter Agent rebased auf aktualisierten main.
Merge-Konflikt	Orchestrator erkennt Konflikt, weist einem Agenten (typischerweise Review Agent) die Konfliktlösung zu. Bei nicht-trivialen Konflikten: Eskalation an menschlichen Entwickler.

Ressourcen-Management

Disk-Budget: Worktrees teilen sich die .git-Datenbank → nur Working Directory wird dupliziert. Bei einem typischen Spring-Boot-Projekt: ~50–200 MB pro Worktree (ohne target/node_modules, die per .gitignore ausgeschlossen sind).

Lebensdauer: Worktrees haben eine maximale TTL (konfigurierbar, Default: 30 Minuten). Nach Ablauf wird der Worktree zwangsweise entfernt und der Task als fehlgeschlagen markiert.

Concurrency-Limit: Maximal n parallele Worktrees pro Repository (konfigurierbar, Default: 5). Verhindert Disk-Exhaustion und zu hohe Build-Last.

Begründung

Warum Git Worktrees statt Feature-Branches mit separatem Clone?

Ein vollständiger `git clone` dupliziert die gesamte Git-History und das Working Directory. Ein Worktree teilt sich die .git-Datenbank mit dem Haupt-Checkout – deutlich schneller zu erstellen (Millisekunden statt Sekunden) und speichereffizienter. Die Isolation ist identisch: Jeder Worktree hat sein eigenes Working Directory und seinen eigenen Branch.

Warum nicht Docker-Container pro Agent?

Docker-Container bieten noch stärkere Isolation (eigenes Dateisystem, eigene Prozesse), aber der Overhead ist erheblich: Image-Pull, Container-Start, Volume-Mounts für den Code, Build-Tool-Installation. Worktrees sind leichtgewichtig (kein OS-Level-Overhead) und direkt in die Git-basierte Orchestrierung (ADR-4) integriert. Docker-Container können bei Bedarf zusätzlich eingesetzt werden (z. B. für Build-Isolation), sind aber kein Ersatz für die Workspace-Isolation.

Warum atomare Merges?

Ein Agent committet und merged entweder alles oder nichts. Partial Merges (nur manche Dateien eines Agents) erzeugen inkonsistente Zustände: ein Service ohne seine Tests, ein Interface ohne seine Implementierung. Die Guardrails-Pipeline (ADR-3) validiert den gesamten Branch – wenn sie besteht, ist der Branch als Ganzes mergebar.

Konsequenzen

Positive Konsequenzen

- Vollständige Isolation: Kein Agent sieht halbfertige Änderungen eines anderen Agenten
- Atomare Merges: Die Guardrails-Pipeline validiert den gesamten Branch, nicht einzelne Commits – Alles-oder-Nichts-Semantik
- Einfaches Rollback: Worktree löschen und Branch entfernen = Änderungen vollständig rückgängig, keine Spuren im Haupt-Branch
- Parallelisierung: Mehrere Agenten arbeiten gleichzeitig ohne Koordination auf Dateisystemebene
- Deterministische Builds: Jeder Agent baut gegen einen definierten Stand (den main-Branch zum Zeitpunkt der Worktree-Erstellung)
- Audit-Trail: Jeder Agent-Branch zeigt exakt, was der Agent geändert hat (sauberer Diff gegen main)

Negative Konsequenzen

- Disk-Overhead: Jeder Worktree dupliziert das Working Directory (~50–200 MB pro Spring-Boot-Projekt). Mitigation: Concurrency-Limit und TTL.
- Merge-Konflikte bei überlappenden Dateien: Wenn zwei Agenten dieselbe Datei ändern, entsteht ein Konflikt. Mitigation: Intelligentes Task-Routing im Orchestrator (überlappende Tasks sequenziell zuweisen) und automatische Rebase-Strategie.
- Veralteter Basis-Stand: Ein lang laufender Agent arbeitet auf einem zunehmend veralteten main. Mitigation: TTL und periodischer Rebase für langlebige Worktrees.
- Build-Redundanz: Jeder Worktree führt eigene Builds aus. Mitigation: Shared Build-Cache (Maven Local Repository, Gradle Build Cache, NX Cache) über alle Worktrees hinweg.

ADR-3: Guardrail Pipeline als Pflicht-Gate

Motivation / Kontext

Large Language Models generieren Code, der syntaktisch korrekt aussieht und funktional plausibel wirkt – aber in Produktionsumgebungen gefährlich sein kann. Die Risiken sind vielfältig:

- **Halluzinationen:** Das Modell ruft APIs auf, die nicht existieren, oder verwendet Bibliotheksversionen mit inkompatiblen Signaturen
- **Security-Lücken:** Generierter Code enthält SQL Injection, unsichere Deserialisierung, hardcodierte Credentials oder fehlende Input-Validierung
- **Architektur-Verstöße:** Ein Agent greift direkt auf die Datenbank eines anderen Bounded Context zu, statt über die definierte Schnittstelle zu kommunizieren
- **Style-Inkonsistenz:** Generierter Code weicht von den Projekt-Konventionen ab (Naming, Package-Struktur, Logging-Pattern)
- **Unzureichende Tests:** Agent generiert Implementierung ohne Tests, oder Tests, die trivial sind (keine Assertions, nur Happy Path)
- **Confidence-Probleme:** Das Modell ist sich unsicher, generiert aber trotzdem Code, anstatt zu eskalieren

Ohne eine automatische, systematische Validierung ist **jeder generierte Code ein Risiko**. Die Guardrails-Pipeline ist die zentrale Qualitätssicherungsmaßnahme der gesamten Agenten-Architektur. Sie stellt sicher, dass kein Code – unabhängig davon, welcher Agent ihn generiert hat – ohne Validierung in die Codebasis gelangt.

Entscheidung

Jede Codeänderung eines Agenten durchläuft eine **sechsstufige Validierungs-Pipeline**. Alle Stufen müssen bestanden werden, bevor ein Merge möglich ist. Bei Fehlschlag erhält der Agent strukturiertes Feedback und kann korrigieren (max. n Retries, danach Eskalation).

Pipeline-Stufen

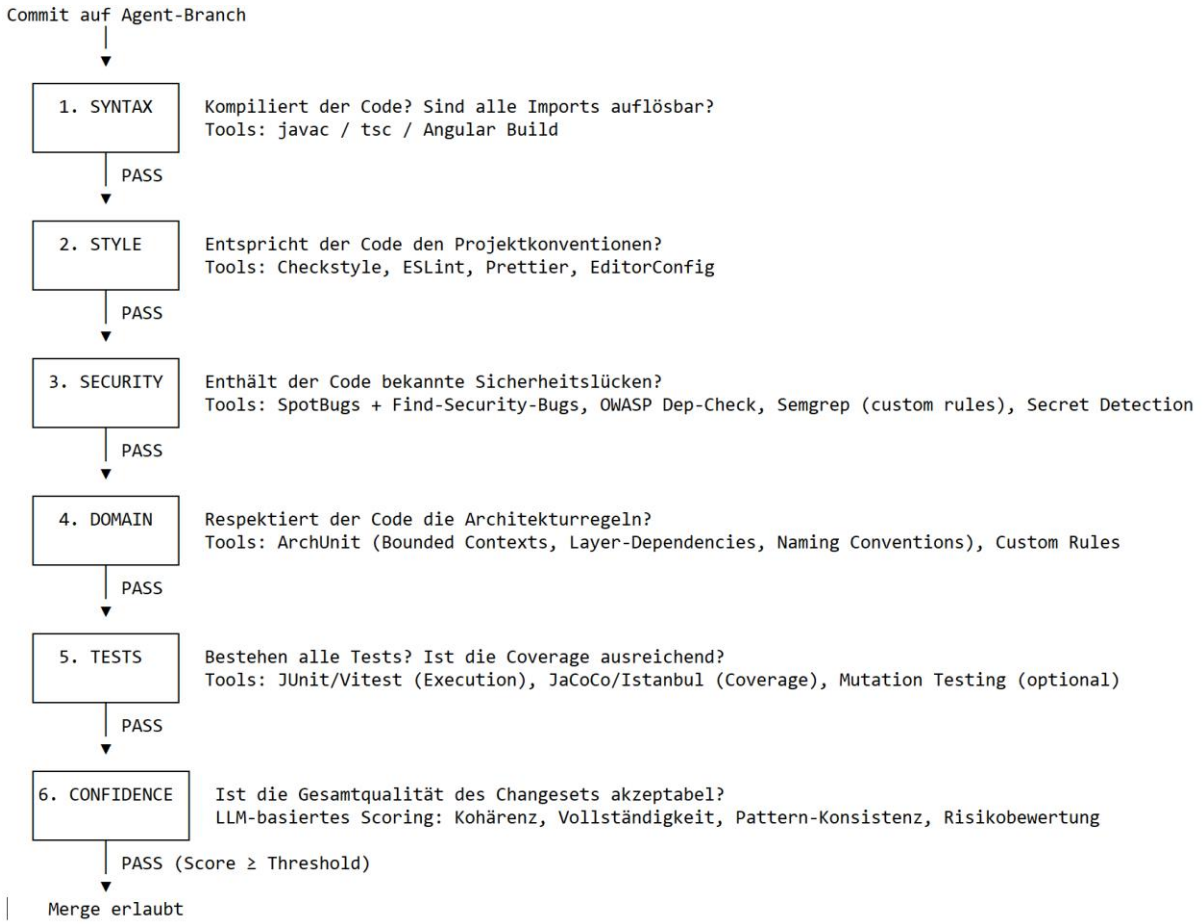


Abbildung 17: Pipeline Stufen

Stufe 1: Syntax-Validierung

Aspekt	Details
Prüfung	Kompilierung (Backend: mvn compile, Frontend: ng build), Dependency Resolution
Typische Fehler	Nicht-existierende Imports, falsche Methodensignaturen, fehlende Dependencies in pom.xml/package.json
Feedback an Agent	Compiler-Fehlermeldungen als strukturierter Text
Ausführungszeit	5–15 Sekunden

Stufe 2: Style-Validierung

Aspekt	Details
Prüfung	Code-Style (Checkstyle, ESLint), Formatierung (Prettier, google-java-format), Naming Conventions
Typische Fehler	Falsche Package-Struktur, inkonsistente Naming-Patterns, fehlende Javadoc auf public APIs
Feedback an Agent	Regel-ID + betroffene Zeile + erwartetes Pattern
Ausführungszeit	3–8 Sekunden

Stufe 3: Security-Validierung

Aspekt	Details
Prüfung	SAST (SpotBugs + Find-Security-Bugs), Dependency-Vulnerabilities (OWASP Dependency-Check), Custom Rules (Semgrep), Secret Detection (Gitleaks)
Typische Fehler	SQL Injection, unsichere Deserialisierung, hardcodierte Credentials, Verwendung vulnerabler Dependencies
Feedback an Agent	CWE-ID, Schweregrad, betroffene Codezeile, Remediation-Hinweis
Ausführungszeit	10–20 Sekunden
Verhalten bei Findings	Critical/High → Pipeline bricht ab. Medium → Warnung, Agent entscheidet. Low → informativ.

Stufe 4: Domain-Validierung

Aspekt	Details
Prüfung	Bounded-Context-Grenzen (kein direkter DB-Zugriff über Kontextgrenzen), Layer-Abhängigkeiten (Domain darf nicht von Infrastructure abhängen), Naming (Aggregate Roots, Value Objects, Repositories)
Tool	ArchUnit mit projektspezifischem Regelset
Typische Fehler	Service A importiert Repository von Service B, Domain-Entity hat Spring-Annotation, Controller enthält Business-Logik
Feedback an Agent	Verletzte Regel + betroffene Klasse + erlaubte Abhängigkeiten laut ADR
Ausführungszeit	5–10 Sekunden

Stufe 5: Test-Validierung

Aspekt	Details
Prüfung	Alle bestehenden Tests bestehen (Regression), neue Tests für neuen Code vorhanden, Coverage ≥ Schwellwert
Tools	JUnit 5 + Vitest (Execution), JaCoCo + Istanbul (Coverage)
Schwellwerte	Line Coverage ≥ 80 % (für den geänderten Code, nicht das Gesamtprojekt)
Typische Fehler	Agent generiert Code ohne Tests, Tests bestehen, prüfen aber nichts (leere Assertions), bestehende Tests brechen
Feedback an Agent	Fehlgeschlagene Tests mit Stack-Trace, Coverage-Delta mit unkovered Lines
Ausführungszeit	10–30 Sekunden (abhängig von Testumfang)

Stufe 6: Confidence Scoring

Aspekt	Details
Prüfung	LLM-basierte Bewertung des gesamten Changesets: Kohärenz (passt der Code zum bestehenden Projekt?), Vollständigkeit (fehlen offensichtliche Teile?), Pattern-Konsistenz (werden etablierte Patterns korrekt angewendet?), Risiko (gibt es subtile Probleme, die die vorherigen Stufen nicht erkennen?)
Modell	Sonnet (kosteneffizient für Scoring) oder Opus (für kritische Changesets)
Scoring	0–100 Score, konfigurierbar Threshold (Default: 70)
Feedback an Agent	Score + Begründung + spezifische Bedenken
Ausführungszeit	5–15 Sekunden
Sonderregel	Score < 50 → automatische Eskalation an menschlichen Reviewer, unabhängig von Retry-Budget

Retry-Mechanismus

Parameter	Wert	Begründung
Max Retries	3	Erfahrungswert: Nach 3 Versuchen ist ein Feedback-Loop ausgeschöpft. Weitere Versuche verbrennen nur Token.
Feedback-Format	Strukturierter JSON mit Stufe, Regel-ID, betroffener Datei/Zeile, Fehlerbeschreibung, Lösungshinweis	Agent kann Feedback direkt als Kontext in den nächsten Versuch einspeisen
Eskalation nach Max Retries	Task wird als „blocked" markiert, menschlicher Reviewer wird benachrichtigt	Verhindert Endlosschleifen und unkontrollierten Token-Verbrauch

Begründung

Warum sechs Stufen statt einer monolithischen Prüfung?

Stufenweise Validierung ermöglicht **Early Exit**: Wenn der Code nicht kompiliert (Stufe 1), ist es sinnlos, Security-Scans oder Tests auszuführen. Jede Stufe gibt spezifisches, actionable Feedback – „Zeile 42: SQL Injection (CWE-89)" ist hilfreicher als „Pipeline failed".

Warum Confidence Scoring als letzte Stufe?

Die ersten fünf Stufen prüfen objektive, regelbasierte Kriterien. Stufe 6 fängt die **subtilen Probleme** ab, die regelbasierte Tools nicht erkennen: Code, der zwar kompiliert und alle Tests besteht, aber konzeptionell falsch ist (z. B. ein Event-Handler, der synchron statt asynchron implementiert ist). Das LLM prüft, ob das Gesamtbild stimmig ist.

Warum kein menschlicher Review als Pflicht?

Die Guardrails-Pipeline ersetzt den menschlichen Review **nicht**, sondern reduziert seinen Aufwand. In der Praxis werden 70–80 % der Agenten-Ausgaben ohne Korrekturbedarf die Pipeline bestehen. Menschliche Reviewer können sich auf die verbleibenden 20–30 % konzentrieren und auf die Confidence-Score-Begründungen als Entscheidungshilfe zugreifen.

Konsequenzen

Positive Konsequenzen

- Systematische, reproduzierbare Qualitätssicherung für jeden generierten Code-Artefakt – unabhängig davon, welcher Agent oder welches Modell den Code generiert hat
- Früherkennung von LLM-Halluzinationen (nicht-existierende APIs, falsche Signaturen) in Stufe 1 (Syntax), bevor aufwendigere Prüfungen starten
- Compliance-Nachweis für Audit-Anforderungen: Jede Pipeline-Ausführung ist protokolliert und nachvollziehbar
- Strukturiertes Feedback ermöglicht Agenten, gezielt zu korrigieren – kein blindes Raten
- Defense in Depth: Sechs unabhängige Prüfebenen – ein Fehler, der Stufe 3 passiert, wird mit hoher Wahrscheinlichkeit in Stufe 4, 5 oder 6 erkannt
- Confidence Scoring fängt subtile, kontextabhängige Probleme ab, die regelbasierte Tools prinzipiell nicht erkennen können

Negative Konsequenzen

- Zusätzliche Laufzeit: 30–90 Sekunden pro Pipeline-Durchlauf (abhängig von Projektgröße und Testumfang). Mitigation: Stufenweises Early Exit und Parallelisierung der Stufen 2–4.
- False Positives (besonders in Stufen 3 und 6) können Agenten-Workflows verlangsamen und Token verschwenden. Mitigation: Regelmäßige Kalibrierung der Regeln und Schwellwerte, Suppress-Mechanismus für bekannte False Positives.
- Initiale Konfiguration der Pipeline erfordert Aufwand: ArchUnit-Regeln für Domain-Validierung, Semgrep-Rules für projektspezifische Security-Patterns, Confidence-Score-Kalibrierung.
- Confidence Scoring (Stufe 6) ist selbst LLM-basiert und damit nicht deterministisch: Derselbe Code kann bei wiederholter Prüfung unterschiedliche Scores erhalten. Mitigation: Score-Schwellwert mit Puffer (70 statt 50).

ADR-4: Git-basierte Orchestrierung

Motivation / Kontext

Die Multi-Agent-Architektur (ADR-1) benötigt einen Orchestrierungsmechanismus, der den Workflow steuert: Welcher Agent arbeitet wann an welcher Aufgabe? Wo werden Zwischenergebnisse abgelegt? Wie wird der Fortschritt nachvollziehbar?

Denkbare Orchestrierungsmechanismen:

Option	Beschreibung	Zusätzliche Infrastruktur
Message Queue (RabbitMQ, Kafka)	Events zwischen Agenten, async Verarbeitung	Broker-Infrastruktur, Monitoring
Workflow-Engine (Temporal, Camunda)	Formale Workflow-Definition, State Management	Engine-Infrastruktur, Deployment
Datenbank-basiert	Shared State in DB, Polling-Mechanismus	Datenbank, Schema-Management
Git-basiert	Branches = States, Commits = Checkpoints, PRs = Gates, Repository = Knowledge Store	Keine – Git ist bereits vorhanden

In einem Agenten-System, das Code generiert, ist Git **bereits der zentrale Artefakt-Speicher**. Jeder Agent liest und schreibt Code im Repository. Der Orchestrierungsmechanismus sollte diesen bestehenden Artefakt-Speicher nutzen, statt einen zweiten, parallelen State-Management-Layer einzuführen.

Entscheidung

Git dient als primärer Orchestrierungsmechanismus.

Branches repräsentieren Workflow-States, Commits sind Checkpoints, Pull Requests (bzw. Merge Requests) sind Review- und Validation-Gates, und ein dediziertes Verzeichnis im Repository (`docs/knowledge/`) dient als Shared Knowledge Store.

Git als State Machine

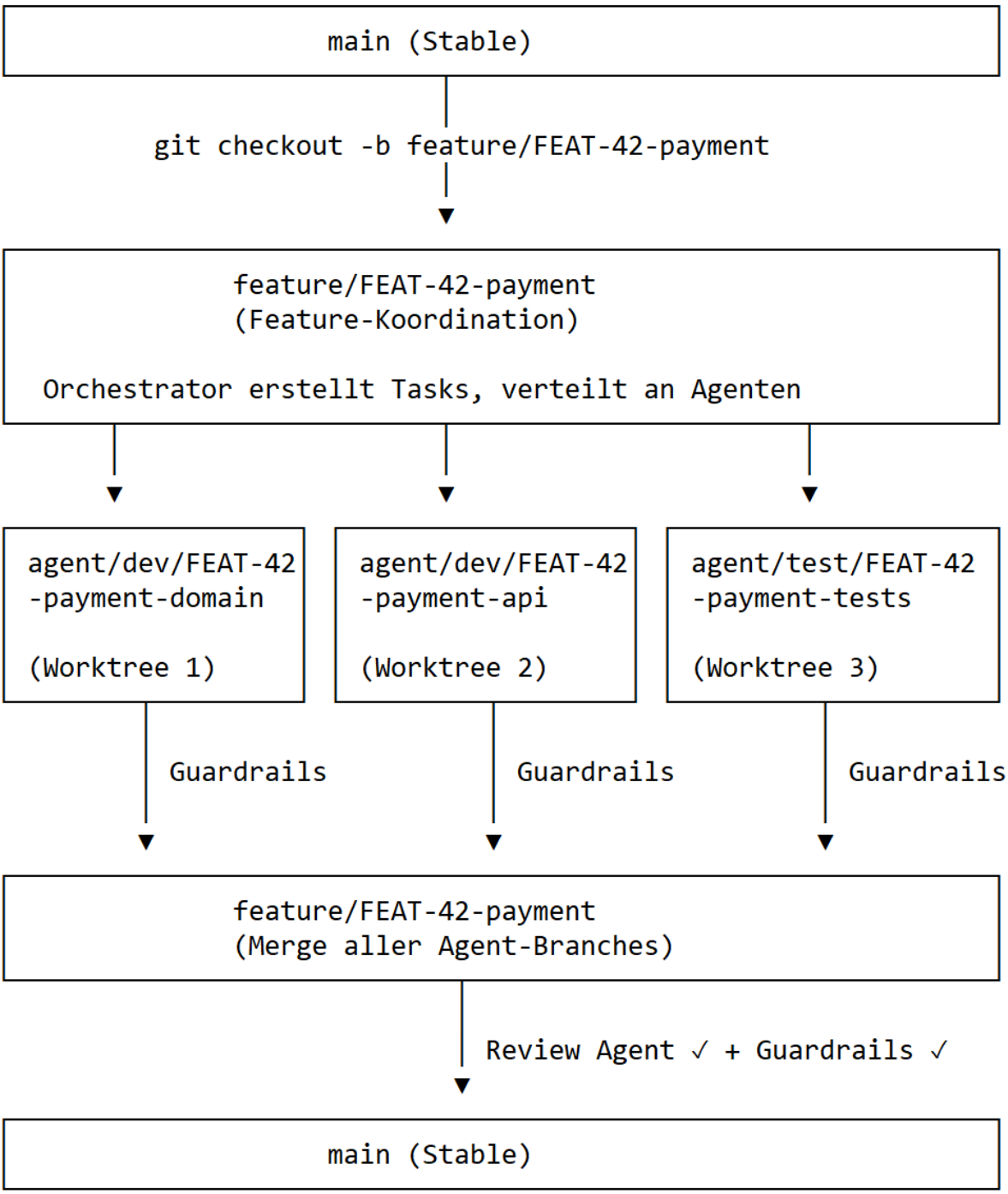


Abbildung 18: Git als State-Machine

Shared Knowledge Store

Das Verzeichnis `docs/knowledge/` im Repository dient als gemeinsamer Wissensspeicher für alle Agenten. Es ist versioniert, durchsuchbar und über Git-History nachvollziehbar.

Verzeichnisstruktur

```
docs/knowledge/
├── architecture/
│   ├── adrs/                      # Architecture Decision Records
│   │   ├── ADR-001-*.md
│   │   └── ...
│   ├── bounded-contexts.md        # DDD Bounded Context Map
│   ├── api-contracts/             # OpenAPI-Specs, AsyncAPI
│   └── patterns.md                # Erlaubte/verbotene Patterns
├── domain/
│   ├── glossary.md                # Ubiquitous Language
│   ├── event-catalog.md           # Domain Events
│   └── aggregates.md              # Aggregate-Beschreibungen
├── conventions/
│   ├── coding-standards.md        # Code-Konventionen
│   ├── testing-standards.md       # Test-Konventionen
│   └── commit-conventions.md      # Commit-Message-Format
├── tasks/
│   ├── active/                    # Laufende Tasks (JSON)
│   │   ├── FEAT-42-task-1.json
│   │   └── FEAT-42-task-2.json
│   ├── completed/                 # Abgeschlossene Tasks
│   └── blocked/                   # Blockierte Tasks
└── memory/
    ├── decisions.md               # Entscheidungen im laufenden
Feature
    ├── lessons-learned.md         # Aus Fehlern gelernt (Retry-
Feedback)
    └── context-cache.md           # Wiederverwendbarer Kontext
```

Task-Datei-Format

```
{
  "id": "FEAT-42-task-1",
  "feature": "FEAT-42-payment",
  "title": "Implement PaymentAggregate with DDD",
  "agent": "development",
  "model": "sonnet",
  "status": "in_progress",
  "worktree": "/tmp/agent-dev-FEAT-42-payment-domain",
  "branch": "agent/dev/FEAT-42-payment-domain",
  "dependencies": [],
  "created_at": "2025-03-04T10:00:00Z",
  "started_at": "2025-03-04T10:00:05Z",
  "guardrails_attempts": 0,
  "max_retries": 3,
  "context_files": [
    "docs/knowledge/architecture/adrs/ADR-001-ddd.md",
    "docs/knowledge/domain/aggregates.md",
    "docs/knowledge/conventions/coding-standards.md"
  ]
}
```

Workflow-Events als Git-Operationen

Workflow-Event	Git-Operation	Nachvollziehbarkeit
Feature gestartet	git checkout -b feature/{id}	Branch-Erstellung in Git-Log
Task an Agent zugewiesen	Task-Datei in docs/knowledge/tasks/active/ committed	Commit-History
Agent startet Arbeit	git worktree add + Branch-Erstellung	Worktree + Branch in Git
Agent erstellt Checkpoint	git commit im Agent-Branch	Commit mit Message
Agent ist fertig	Task-Datei → completed/, PR gegen Feature-Branch	PR-History
Guardrails bestanden	Merge des Agent-Branch in Feature-Branch	Merge-Commit
Feature abgeschlossen	PR von Feature-Branch gegen main	PR + Review-History

Vorteile gegenüber externen Orchestrierungstools

Aspekt	Git-basiert	Externe Tools (Temporal, Kafka, etc.)
Zusätzliche Infrastruktur	Keine	Broker/Engine muss betrieben werden
Nachvollziehbarkeit	Git-History ist der Audit-Trail	Separater Log-/Event-Store nötig
CI/CD-Integration	Nativ (PRs triggern Pipelines)	Adapter/Webhooks nötig
Persistenz	Git-Repository = persistent by default	State-Store muss gesichert werden
Wiederherstellung	git log + git reset = vollständiger Zustand	Tool-spezifische Recovery-Mechanismen
Lernkurve	Git kennt jeder Entwickler	Tool-spezifisches Wissen nötig

Begründung

Warum Git statt Message Queue?

Message Queues (RabbitMQ, Kafka) sind für lose gekoppelte, asynchrone Systeme mit hohem Durchsatz optimiert. Agenten-Workflows sind aber **koordiniert**, nicht lose gekoppelt: Der Orchestrator muss wissen, welche Tasks abgeschlossen sind, bevor er die nächsten startet. Git-Branches und PRs bilden diese Abhängigkeiten natürlicher ab als Events in einer Queue.

Warum Git statt Workflow-Engine?

Workflow-Engines (Temporal, Camunda) bieten formale State Machines, Retry-Mechanismen und Monitoring. Der Overhead ist jedoch erheblich: Engine-Deployment, Workflow-Definition, Worker-Registrierung. Für ein Agenten-System mit 5–10 parallelen Agenten und klar definierten Phasen (Plan → Develop → Test → Review → Merge) ist Git ausreichend. Wenn das System auf 50+ Agenten skaliert, sollte diese Entscheidung revisited werden.

Warum der Knowledge Store im Repository und nicht in einer Datenbank?

Der Knowledge Store ist primär Textdokumente: ADRs, API-Specs, Konventionen, Glossare. Diese sind in Markdown/JSON im Repository natürlicher zuhause als in einer Datenbank. Agenten lesen diese Dateien als Teil ihres Kontextfensters – ein File-Read ist einfacher als

ein DB-Query. Zudem ist der Knowledge Store damit automatisch versioniert und über PRs reviewbar.

Konsequenzen

Positive Konsequenzen

- Kein zusätzlicher Infrastrukturbedarf: Git-Repository ist bereits vorhanden, kein Broker, keine Engine, keine Datenbank
- Vollständige Nachvollziehbarkeit: Jeder Workflow-Schritt ist ein Git-Commit, jede Entscheidung ein Datei-Eintrag im Knowledge Store – die gesamte Historie ist über `git log` abrufbar
- Nahtlose CI/CD-Integration: Pull Requests gegen Feature-Branches triggern automatisch die Guardrails-Pipeline – kein zusätzliches Webhook-Setup
- Versionierter Knowledge Store: Architekturentscheidungen, Domain-Wissen und Konventionen sind im selben Repository wie der Code – immer synchron, immer reviewbar
- Entwickler-freundlich: Jeder Entwickler versteht Git-Branches, PRs und Merges – keine Einarbeitung in ein neues Orchestrierungstool
- Offline-fähig: Git funktioniert lokal, kein Netzwerkzugang zu externen Services nötig

Negative Konsequenzen

- Git-Operationen können bei großen Repositories (>1 GB, >100.000 Commits) langsam werden. Mitigation: Shallow Clones für Worktrees, regelmäßiges Repository-Housekeeping, Git LFS für große Dateien.
- Shared Knowledge Store ist dateibasiert: Kein Index, keine Query-Language, keine Transaktionen. Mitigation: Klare Verzeichnisstruktur, JSON für maschinenlesbare Daten, Markdown für menschenlesbare Dokumente.
- Skalierungsgrenzen bei >10 parallelen Agenten: Viele gleichzeitige Branches, häufige Merges, potenzielle Git-Lock-Contention. Mitigation: Concurrency-Limit (ADR-2), sequenzielle Merges für überlappende Dateien.
- Task-Status-Management über Dateien in `docs/knowledge/tasks/` ist rudimentär: Kein Dashboard, kein Alerting, keine automatische Timeout-Erkennung. Mitigation: Leichtgewichtiges CLI-Tool, das Task-Dateien auswertet und Statusübersichten generiert.
- Keine native Unterstützung für komplexe Workflow-Patterns (Compensating Transactions, Saga Pattern). Für einfache Plan→Develop→Test→Review→Merge-Workflows ausreichend, bei komplexen Orchestrierungen revisit nötig.

20. Vergleich: Claude Code vs. andere KI-Entwicklungssysteme

Der Markt für KI-gestützte Entwicklungswerkzeuge wächst rasant. Die folgende Gegenüberstellung ordnet Claude Code in das Ökosystem ein und verdeutlicht die architektonischen Unterschiede:

Kriterium	Claude Code	Devin (Cognition)	Cursor / Copilot
Architektur-Modell	Orchestrierte Multi-Agent-Pipeline	Autonomer Single-Agent	IDE-integrierter Assistent
Autonomiegrad	Gesteuert: Human-in-the-Loop an definierten Gates (AP-6)	Hochautonom: Agent arbeitet weitgehend selbstständig	Niedrig: Entwickler steuert, KI assistiert
Anpassbarkeit	Hoch: CLAUDE.md, Custom Agents, Hooks, MCP-Server	Begrenzt: Vordefinierte Capabilities	Mittel: Prompt-Konfiguration, Extensions
Enterprise-Governance	Eingebaut: Guardrails, Audit-Trail, Least Privilege (AP-3)	Begrenzt: Keine formale Guardrails-Pipeline	Nicht vorhanden: IDE-Tool ohne Governance
DDD-Integration	Native Unterstützung: Bounded Contexts, Glossar, Domain Hooks	Nicht vorhanden	Nicht vorhanden
Kostenmodell	Transparent: Token Budgets, Modellwahl pro Agent	Subscription-basiert (Fixkosten)	Subscription-basiert (Fixkosten)
Offline/Air-Gap	Ja: Lokal mit Ollama/Weaviate möglich	Nein: Cloud-only	Teilweise: Copilot nur Cloud, Cursor hybrid
CI/CD-Integration	Nativ über CLI, Hooks und Git-basierte Orchestrierung	Eigenständige Plattform	Nur IDE-seitig

Claude Code positioniert sich als das Enterprise-tauglichste System: Es bietet die höchste Anpassbarkeit, eingebaute Governance und die Möglichkeit, Agenten über CLAUDE.md deklarativ auf Projektstandards zu konfigurieren. Devin ist die autonomste Lösung, aber mit weniger Kontrollmöglichkeiten. Cursor/Copilot sind die niedrigschwelligsten Werkzeuge, aber für Enterprise-Workflows mit Multi-Agent-Orchestrierung nicht ausgelegt.

21. End-to-End: Payment Service Implementation

Während die vorherigen Kapitel die Architektur, das Ausführungsmodell und die Governance-Mechanismen eines agentischen Entwicklungssystems beschreiben, zeigt dieses Kapitel den vollständigen Ablauf einer realistischen Feature-Implementierung.

Als Beispiel dient die Entwicklung eines **Payment Services** für eine E-Commerce-Plattform. Der Service stellt eine REST-API zur Verarbeitung von Zahlungen bereit, publiziert Domain Events über Kafka und wird containerisiert in einer Kubernetes-Umgebung betrieben.

Der Ablauf demonstriert, wie der Orchestrator spezialisierte Agenten entlang des Software Development Lifecycle koordiniert. Beginnend mit der Anforderungsanalyse werden Architekturentscheidungen getroffen, Implementierungsaufgaben geplant, Code generiert, Tests erstellt und schließlich das Deployment vorbereitet.

Dabei wird sichtbar, wie die zuvor eingeführten Konzepte – insbesondere **Task Graph, Workspace Isolation, Guardrails Pipeline und Shared Knowledge Store** – in einem zusammenhängenden Workflow zusammenspielen.

Das folgende Beispiel zeigt die agentische Orchestrierung eines solchen Entwicklungsauftrags in Form eines vereinfachten Task-Workflows.

21.1 Ausgangssituation

In diesem Szenario wird die Implementierung eines Payment Services für eine E-Commerce-Plattform betrachtet. Der Service stellt eine REST-basierte API zur Verarbeitung von Zahlungen bereit, verwaltet Zahlungszustände innerhalb eines Domain-Driven-Design-Modells und publiziert relevante Domain Events über Apache Kafka, um andere Systeme über erfolgreiche oder fehlgeschlagene Transaktionen zu informieren.

Der Service muss dabei mehrere Anforderungen erfüllen. Dazu gehören eine klare Trennung der Domänenschichten gemäß hexagonaler Architektur, die Einhaltung regulatorischer Anforderungen wie PSD2 sowie eine sichere Integration in bestehende Plattformkomponenten. Darüber hinaus muss der Service containerisiert bereitgestellt und in einer Kubernetes-Umgebung betrieben werden können.

Der Entwicklungsauftrag wird nicht manuell umgesetzt, sondern durch ein agentisches Entwicklungssystem orchestriert. Ein zentraler Orchestrator übersetzt das Entwicklungsziel in einen Task Graph, der von spezialisierten Agenten entlang des Software Development Lifecycle abgearbeitet wird. Jeder Agent übernimmt dabei eine klar abgegrenzte Verantwortung, beispielsweise für Anforderungsanalyse, Architekturentscheidungen, Implementierung, Tests oder Deployment.

Während der gesamten Ausführung greifen die Agenten auf einen Shared Knowledge Store zu, in dem Projektdokumentation, Architekturentscheidungen und relevante Kontextinformationen abgelegt sind. Gleichzeitig stellt eine mehrstufige Guardrails-Pipeline sicher, dass generierter Code Qualitäts-, Sicherheits- und Architekturregeln einhält.

Das folgende Beispiel zeigt, wie ein solcher Entwicklungsauftrag durch den Orchestrator in mehrere Phasen zerlegt und durch spezialisierte Agenten umgesetzt wird.

21.2 Gesamtworkflow

Der folgende Ablauf zeigt den vollständigen Lebenszyklus einer Feature-Implementierung innerhalb des agentischen Entwicklungssystems. Ein Feature-Request wird vom Orchestrator entgegengenommen und in mehrere Phasen des Software Development Lifecycle zerlegt. Für jede Phase wird ein spezialisierter Agent gestartet, der innerhalb eines isolierten Workspaces arbeitet und auf den gemeinsamen Wissensspeicher zugreifen kann.

Der Workflow beginnt mit der Analyse der Anforderungen und der Ableitung einer geeigneten Architektur. Anschließend erstellt der Planungsagent einen Implementierungsplan, der von mehreren Entwicklungsagenten umgesetzt wird. Nach der Implementierung werden automatisch Tests generiert und ausgeführt, bevor ein Review-Agent die Einhaltung von Architektur- und Sicherheitsregeln prüft.

Abschließend übernimmt ein Deployment-Agent die Erstellung der notwendigen Infrastrukturartefakte und bereitet das Deployment in der Zielumgebung vor.

Die folgende Abbildung zeigt den Gesamtworkflow der agentischen Umsetzung eines Features.

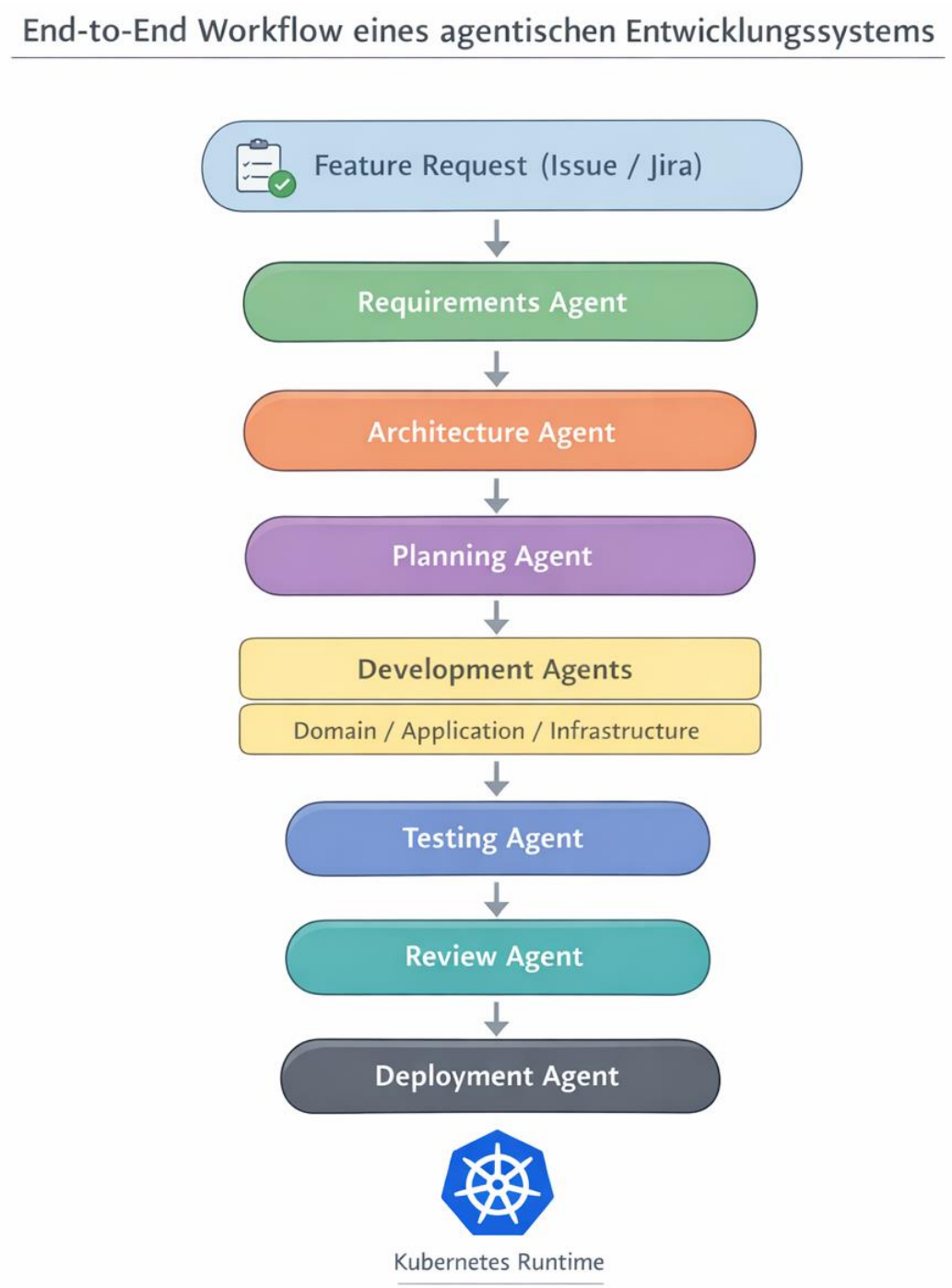


Abbildung 19 End-to-End Workflow

Während das vorherige Diagramm den Ablauf eines Entwicklungsauftrags zeigt, ist für das Verständnis der Architektur auch relevant, welche Systemkomponenten während dieses Prozesses miteinander interagieren.

Der Orchestrator fungiert dabei als zentrale Steuerungseinheit des agentischen Entwicklungssystems. Er übersetzt Entwicklungsziele in einen Task Graph und startet spezialisierte Agenten, die jeweils klar abgegrenzte Aufgaben übernehmen. Diese Agenten arbeiten in isolierten Workspaces und greifen über definierte Tool-Schnittstellen auf Build-Systeme, Testframeworks und Deployment-Werkzeuge zu.

Die folgende Abbildung zeigt, wie die einzelnen Architekturkomponenten während eines End-to-End-Workflows zusammenwirken.

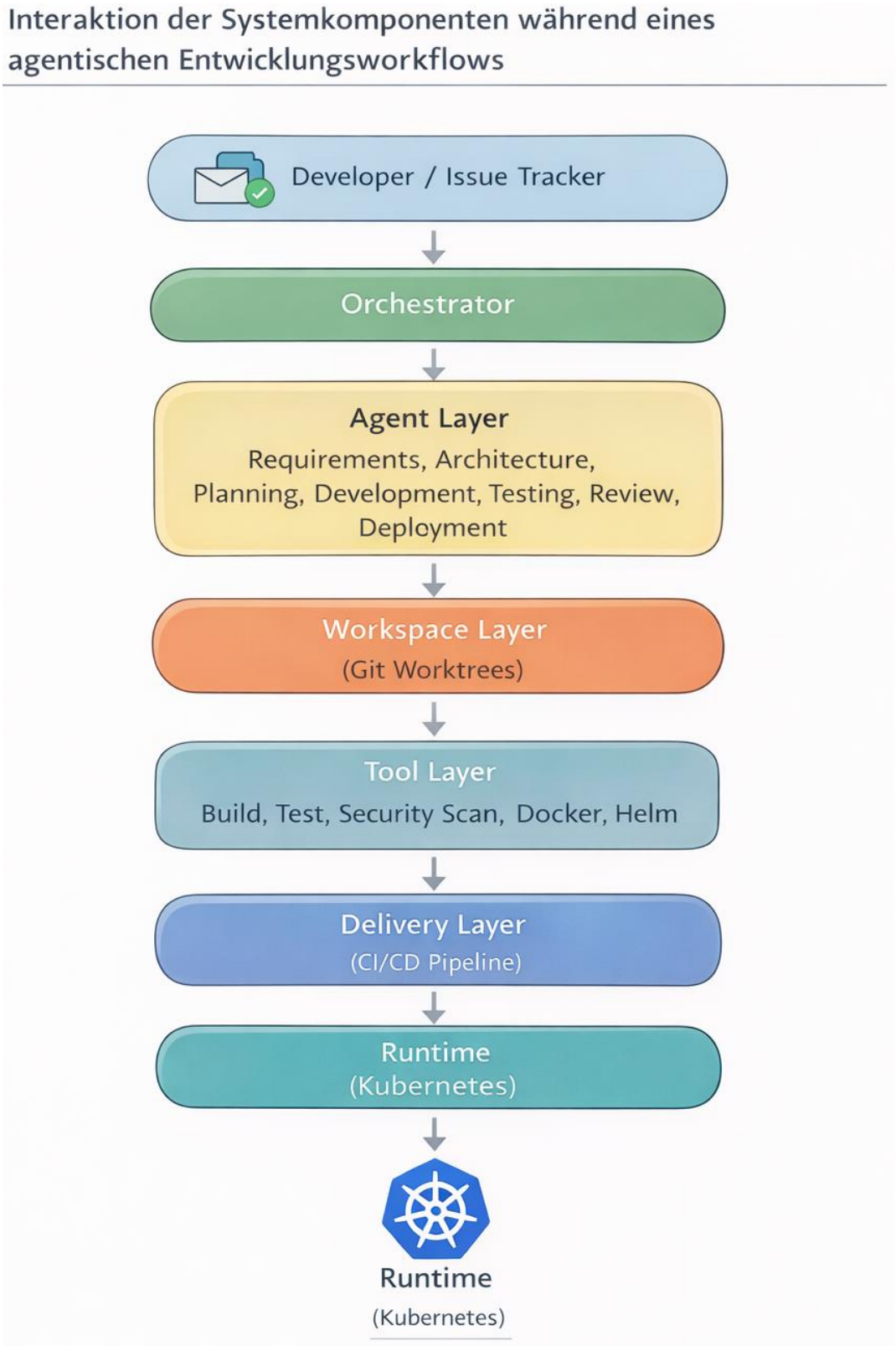


Abbildung 20 Interaktion der Systemkomponenten

21.3 Agentischer Workflow

```
// PHASE 1: Requirements & Architektur
Task(subagent_type="requirements-agent",
    prompt="Stories für payment-service aus Jira Sprint 43. RTM erstellen.")
Task(subagent_type="architecture-agent", model="opus",
    prompt="Hexagonale Architektur, DDD, Kafka, PSD2. Output: ADR + Shared Knowledge Store")

// PHASE 2: Planung
Task(subagent_type="planning-agent",
    prompt="Implementierungstracker: Domain -> Application -> Infra -> Test -> Deploy")

// PHASE 3: Implementierung (sequenziell + parallel)
Task(subagent_type="dev-agent", prompt="Domain Layer: Entities, Value Objects, Events.")
Task(subagent_type="dev-agent", prompt="Application: Controllers, Security.", isolation="worktree")
Task(subagent_type="dev-agent", prompt="Infra: Kafka, Outbox.", isolation="worktree")

// PHASE 4: Testing & Review
Task(subagent_type="test-agent", prompt="JUnit 5 + Testcontainers. Min. 80% Coverage.")
Task(subagent_type="review-agent", model="opus", prompt="PSD2, Confidence, Domain-Compliance.")

// PHASE 5: Deployment
Task(subagent_type="deploy-agent", prompt="K8s-Manifeste, HPA, TLS-Ingress.")
Task(subagent_type="qa-guard", prompt="Finale Validierung: 100% Tests, Security.")
```

21.4 Artefakte des Workflows

Der agentische Workflow erzeugt nicht nur Quellcode, sondern eine Reihe strukturierter Artefakte, die den gesamten Entwicklungsprozess nachvollziehbar und wiederverwendbar machen. Dazu gehören Anforderungen, Architekturentscheidungen, Implementierungspläne, Quellcode, Tests sowie Deployment-Artefakte.

Im vorliegenden Beispiel entstehen typischerweise folgende Ergebnisse:

- Anforderungsdokumentation und Traceability Matrix für den Payment Service
- Architekturartefakte wie ADRs und Einträge im Shared Knowledge Store
- Implementierter Code für Domain-, Application- und Infrastructure-Layer
- Automatisch erzeugte Unit- und Integrationstests
- Deployment-Artefakte wie Kubernetes-Manifeste, HPA-Konfiguration und TLS-Ingress-Regeln

Diese Artefakte bilden zusammen die technische und fachliche Grundlage für den produktiven Betrieb des Payment Services. Gleichzeitig ermöglichen sie eine vollständige Nachvollziehbarkeit der Entscheidungen und Änderungen entlang des gesamten Entwicklungslebenszyklus.

21.5 Guardrails Pipeline

Ein wesentliches Merkmal des End-to-End-Szenarios ist die konsequente Einbettung von Guardrails in jede Phase des Entwicklungsprozesses. Agentisch erzeugte Änderungen werden nicht ungeprüft übernommen, sondern durchlaufen eine mehrstufige Validierungs- und Governance-Pipeline.

Im Payment-Service-Beispiel umfasst diese Pipeline insbesondere:

- Syntax- und Compile-Prüfungen zur Sicherstellung der technischen Korrektheit
- Style- und Konventionsprüfungen gemäß den in CLAUDE.md definierten Projektstandards
- Security-Scans zur Erkennung potenzieller Schwachstellen oder problematischer Abhängigkeiten

- Domain-Prüfungen zur Absicherung von Bounded Contexts, Glossarregeln und DDD-Konventionen
- Automatisierte Tests mit vorgegebenen Mindestanforderungen an die Testabdeckung
- Confidence-Scoring zur Bewertung unsicherer oder potenziell halluzinierter Änderungen

Erst wenn alle Prüfungen erfolgreich bestanden wurden, kann der Workflow in die nächste Phase übergehen. Dadurch wird sichergestellt, dass agentisch erzeugter Code denselben Qualitäts-, Sicherheits- und Architekturmaßstäben genügt wie manuell entwickelte Software.

21.6 Deployment Ergebnis

Das End-to-End-Szenario zeigt, dass ein agentisches Entwicklungssystem weit mehr ist als ein Werkzeug zur Codegenerierung. Der Orchestrator koordiniert spezialisierte Agenten entlang des gesamten Software Development Lifecycle und verbindet Anforderungen, Architektur, Implementierung, Tests, Review und Deployment in einem konsistenten Ablauf.

Im Payment-Service-Beispiel wird deutlich, dass insbesondere folgende Vorteile erzielt werden können:

- klare Trennung von Verantwortlichkeiten zwischen spezialisierten Agenten
- reproduzierbare und nachvollziehbare Entwicklungsabläufe
- frühzeitige Validierung von Architektur-, Sicherheits- und Qualitätsanforderungen
- bessere Wiederverwendbarkeit von Wissen durch Shared Knowledge Store und ADRs
- höhere Geschwindigkeit bei gleichzeitig kontrollierter Qualitätssicherung

Damit wird das agentische Entwicklungssystem zu einer belastbaren Architektur für die Umsetzung komplexer Features in Enterprise-Umgebungen. Das Beispiel des Payment Services verdeutlicht, wie die in den vorangegangenen Kapiteln beschriebenen Konzepte in der Praxis zusammenspielen und gemeinsam einen produktionsnahen Entwicklungsprozess ermöglichen.

22. Troubleshooting & Schnellreferenz

Problem	Lösung
Unvollständige Ergebnisse	Prompts aufteilen. max_turns erhöhen.
Falsche Dateien geändert	Pfade explizit angeben. isolation="worktree".
Kontextverlust	resume mit agent_id nutzen.
Parallele Kollisionen	Worktree-Isolation für alle parallelen Agenten.
Halluzinierte APIs	Guardrail-Hooks aktivieren. opus für kritische Tasks.
Domain-Verstöße	glossary.md aktualisieren. Domain-Hook implementieren.
Budget überschritten	Token Budget Tracker einsetzen. Modell-Eskalation prüfen.
MCP-Server Fehler	Umgebungsvariablen und Netzwerkzugriff prüfen.

Aufgabe	Agent
Codebasis analysieren	Explore / architecture-agent
Implementierungsplan erstellen	planning-agent
Anforderungen sammeln	requirements-agent
Features implementieren	dev-agent (sonnet)
Tests schreiben/ausführen	test-agent
Code-Qualität prüfen	review-agent (opus)
Deployment	deploy-agent
CI/CD Quality Gate	qa-guard

23. Glossar & Abkürzungsverzeichnis

Abkürzung / Begriff	Erklärung
ADR	Architecture Decision Record – Dokumentierte Architekturentscheidung.
AOP	Aspect-Oriented Programming – Querschnittsbelange (Logging, Security).
AP-1 bis AP-6	Die sechs architektonischen Prinzipien des Agentensystems (Kapitel 2).
BPMN	Business Process Model and Notation – Geschäftsprozessmodellierung.
CI/CD	Continuous Integration / Deployment – Automatisierte Pipeline.
DAG	Directed Acyclic Graph – Gerichteter azyklischer Graph (Task Graph).
DDD	Domain-Driven Design – Fachliche Softwaremodellierung.
DTO	Data Transfer Object – Datenobjekt ohne Geschäftslogik.
HPA	Horizontal Pod Autoscaler – Kubernetes-Skalierung.
IaC	Infrastructure as Code – Deklarative Infrastruktur.
JPA	Jakarta Persistence API – ORM-Standard für Java.
JWT	JSON Web Token – Authentifizierungstoken.
K8s	Kubernetes – Container-Orchestrierung.
LLM	Large Language Model – Großes Sprachmodell (z.B. Claude).
MCP	Model Context Protocol – Anthropics Tool-Erweiterungsprotokoll.
OWASP	Open Web Application Security Project – Sicherheitsstandards.
PSD2	Payment Services Directive 2 – EU-Zahlungsdiensterichtlinie.
RBAC	Role-Based Access Control – Rollenbasierte Zugriffskontrolle.
RFC 7807	Problem Details for HTTP APIs – Strukturierte Fehlerantworten.
RTM	Requirements Traceability Matrix – Anforderungsverfolgung.
SBOM	Software Bill of Materials – Software-Komponentenliste.
SDLC	Software Development Lifecycle – Entwicklungslebenszyklus.
Sealed Interface	Java-Feature zur Einschränkung implementierender Klassen.
Testcontainers	Docker-basierte Integrationstests für Java.