Agentic Software Development

Enterprise Architecture with AI Agents

Von kontrollierten Einzel-Runs zur parallelen Agentic Software Factory. Praxisbeispiele mit Java, Spring Boot und Domain-Driven Design

Whitepaper

Martin Janda

martin@janda.io

Version 3.0 · **Stand: 10. August 2026**

Inhaltsverzeichnis

Abbildungsverzeichnis

Management Summary

KI-gestützte Coding-Agenten können Anforderungen analysieren, Code verändern, Tests ausführen und Entwicklungsartefakte erzeugen. Ihr Einsatz in Enterprise- und regulierten Umgebungen erfordert jedoch mehr als ein leistungsfähiges Modell: Agentenarbeit muss spezifiziert, begrenzt, isoliert, geprüft, freigegeben und nachträglich rekonstruiert werden können.

Dieses Whitepaper beschreibt eine implementierte Control-Plane-Architektur für diesen Zweck. Die Referenzimplementierung SoftwareFabrik steuert zustandsbehaftete Agentenläufe, projiziert versionierte Regeln in den Workspace, begrenzt Modelle und Budgets, isoliert Ausführungen, bewertet Änderungen durch unabhängige Reviewer und führt relevante Entscheidungen in einem attestierten Audit- und Warum-Trace zusammen.

Der aktuelle Implementierungsstand verwendet genau einen schreibenden Agenten je Run. Mehrere read-only Reviewer prüfen den entstandenen Diff parallel. Diese Architektur vermeidet unkontrollierte Schreibkonkurrenz und bildet die Basis für den nächsten Entwicklungsschritt.

Die Weiterentwicklung ergänzt eine übergeordnete Workflow-Ebene. Ein Feature wird in einen Task Graph zerlegt; voneinander unabhängige Tasks können als isolierte Child Runs parallel ausgeführt werden. Jeder Child Run besitzt einen eigenen Branch oder Worktree und bleibt dem Single-Writer-Prinzip unterworfen. Ein Merge Coordinator und ein abschließendes Integration Gate führen die Ergebnisse kontrolliert zusammen.

Diese Ebene ist vollständig umgesetzt (Stand: SoftwareFabrik-Release 0.30.0, August 2026) — hinter einem standardmäßig deaktivierten Feature-Flag (zwei Detailmessungen wirken auch ohne — 19.10). Der Weg dorthin folgte bewusst der Risikoreihenfolge: erst parallele Analyse ohne Schreibrechte, dann Schreibrechte unter einer Besitzregel, dann versionierte Verträge, begründungspflichtiges Umplanen, eine Merge-Diagnose, eine Koordinationsschicht — begleitet von einer Messung, die ausweist, was sie nicht messen kann, statt es zu schätzen. Welches Release welchen Schritt brachte, dokumentiert die Release-Verlaufstabelle in Kapitel 19.10. Der verteilte Betrieb über mehrere Hosts bleibt bewusst zurückgestellt, solange der Bedarf nicht gemessen ist.

Das Whitepaper unterscheidet konsequent zwischen implementiertem Stand, Zielarchitektur und Roadmap. Quantifizierte Produktivitäts- und ROI-Werte werden als Hypothesen beziehungsweise Modellrechnungen behandelt, solange keine kontrollierte Vergleichsmessung vorliegt.

Kernaussagen

- **Control Plane statt unkontrollierter Autonomie:** Der entscheidende Schritt ist nicht die maximale Zahl gleichzeitig arbeitender Agenten, sondern eine Steuerschicht, die nichtdeterministische Agentenarbeit begrenzt, isoliert, koordiniert, prüft und nachweisbar macht.

- **Single Writer, Multiple Reviewers:** Innerhalb eines Workspace schreibt genau ein Agent; mehrere unabhängige Reviewer prüfen read-only.
- **Parallelität nach Abhängigkeiten:** Parallelität entsteht zwischen isolierten Tasks und Child Runs — abgeleitet aus Abhängigkeiten, Verträgen und Schreibbereichen, nicht aus festen Agentenrollen.
- **Governance by Design:** Governance ist keine Ergänzung der Agentenarchitektur, sondern ein Teil ihrer Struktur — von der Policy-Prüfung bis zur signierten Audit-Hashkette.
- **Git plus autoritativer Prozesszustand:** Git trägt Artefakte, Branches und Checkpoints; Workflow-, Policy-, Freigabe- und Audit-Zustand liegen autoritativ in der Datenbank.
- **Regelkreis statt blindem Retry:** Wiederholungen sind nur zulässig, wenn neue Informationen eingehen — Build-Ausgaben, Reviewer-Findings, Merge-Konflikte, CI-Ergebnisse.
- **Souveränität vor Infrastrukturkomplexität:** Die Architektur bleibt lokal, on-premises und ohne Cloud-Abhängigkeit betreibbar — bis hin zum Air-Gap-Betrieb.

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, Spring Boot und bewährte Enterprise-Patterns – die Prinzipien sind jedoch technologieunabhängig übertragbar.

Teil I.

Grundlagen & Architektur

1. Warum Coding-Agenten eine Prozessschicht brauchen

Coding-Agenten können bestehende Repositories analysieren, Änderungen vornehmen, Build- und Testwerkzeuge aufrufen und Ergebnisse iterativ korrigieren. Ihre Fähigkeit zur selbstständigen Werkzeugnutzung unterscheidet sie von klassischen Codegeneratoren, macht ihre Ausführung aber zugleich nichtdeterministisch und sicherheitsrelevant. Für den Enterprise-Einsatz genügt daher nicht die Auswahl eines leistungsfähigen Modells. Erforderlich ist eine Prozess- und Governance-Schicht, die Agentenarbeit spezifiziert, begrenzt, isoliert, prüft und nachweisbar macht.

Diese Schicht ist der Gegenstand dieses Whitepapers. Es beschreibt eine Control-Plane-Architektur für agentische Softwareentwicklung — vendor-neutral: Der konkrete Coding-Agent (etwa Claude Code, die Codex-CLI oder die Gemini CLI; Stand August 2026, vgl. Kapitel 21) ist darin eine austauschbare Ausführungskomponente hinter einer Schnittstelle, nicht das Fundament der Architektur. Wo dieses Dokument Werkzeugdetails zeigt, dient Claude Code als durchgängiges Beispiel; die Prinzipien gelten werkzeugübergreifend.

Der Fokus liegt auf drei Kernaspekten: Erstens einem geteilten, versionierten Wissensstand, damit Erkenntnisse aus der Analyse nachvollziehbar in die Implementierung einfließen. Zweitens einer fachlichen Modellierung nach DDD-Prinzipien, die den erzeugten Code an fachliche Grenzen bindet. Drittens einer unabhängigen Prüfschicht mit Gates, die strukturierte Befunde liefert — einschließlich der Prüfung der Behauptungen eines Agenten über seine eigene Arbeit (Claim Verification) —, bevor Änderungen in die Codebasis gelangen.

Das Orchestrator-Prinzip

Das Herzstück der Architektur ist das Orchestrator-Prinzip: Eine zentrale Steuerungsinstanz — vergleichbar mit einem Technical Lead — verteilt Aufgaben, überwacht Fortschritte, besitzt den Prozesszustand und führt Ergebnisse zusammen. In Claude Code übernimmt diese Rolle beispielsweise die Hauptsitzung, die spezialisierte Subagenten mit jeweils eigenem Kontextfenster startet; in der in Kapitel 19 beschriebenen Referenzimplementierung ist es ein Orchestrierungsdienst, der als einzige Stelle Statusübergänge ausführt — und damit zugleich die Stelle, an der Governance überhaupt ansetzen kann (19.3, 19.8).

Die Kontextisolation der Agenten ist dabei ein echter Vorteil — sie verhindert Interferenzen zwischen Aufgaben —, aber sie ist keine Workspace- oder Sicherheitsisolation: Wer schreiben darf, wo geschrieben wird und was als geprüft gilt, muss die Prozessschicht regeln (Kapitel 2, 12, 14). Agenten tauschen Informationen über einen geteilten, versionierten Wissensstand aus, ohne ihre Kontextisolation zu durchbrechen.

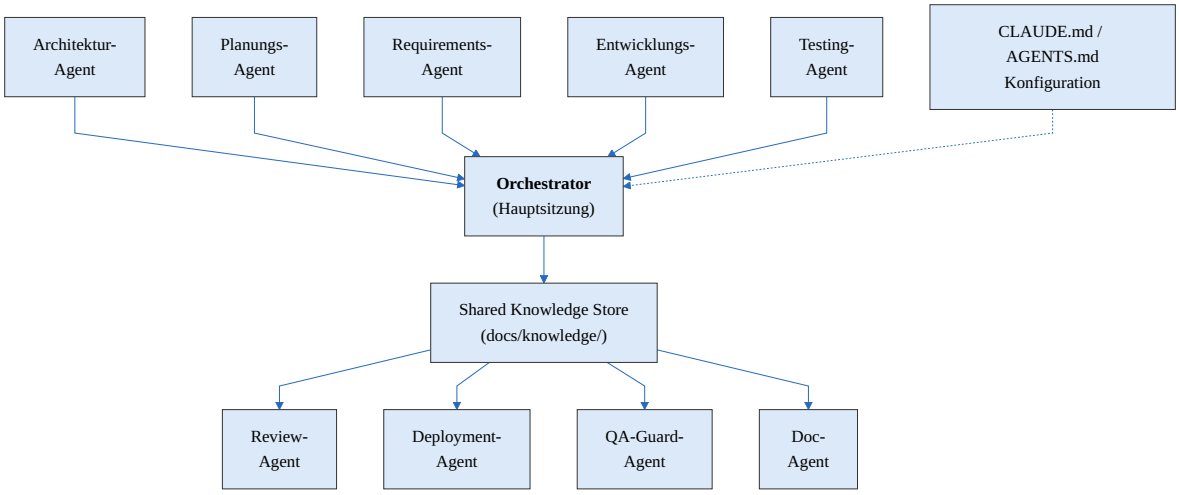

Abbildung 1.1.: Hub-and-Spoke-Multi-Agent-Architektur mit zentralem Orchestrator

Kernprinzipien des Orchestrator-Modells

Prinzip	Beschreibung
Separation of Concerns	Jeder Agent hat eine klar definierte Rolle und einen begrenzten Werkzeugzugriff.
Kontrollierte Parallelität	Unabhängige Aufgaben können gleichzeitig laufen — sofern Abhängigkeiten und Schreibbereiche es erlauben (Kapitel 2).
Autonome Operation	Agenten arbeiten eigenständig innerhalb ihres definierten Rahmens.
Ergebnis-Aggregation	Der Orchestrator sammelt und synthetisiert die Ergebnisse.
Konfigurierbarkeit	Regeln und Agentendefinitionen liegen deklarativ und versioniert im Repository (AGENTS.md/CLAUDE.md, Kapitel 4).
Wiederaufnahme	Läufe sind zustandsbehaftet und können fortgesetzt werden.
Shared State	Ein gemeinsamer, versionierter Wissensstand ermöglicht Zusammenarbeit.
Unabhängige Prüfung	Read-only-Reviewer und Gates liefern strukturierte Befunde — einschließlich Claim Verification — und geben Änderungen frei oder blockieren sie.

2. Architektonische Prinzipien

Jedes Enterprise-Architekturdokument basiert auf einem Set formaler Prinzipien, die als Leitplanken für alle Designentscheidungen dienen. Die folgenden acht Prinzipien bilden das Fundament der in diesem Whitepaper beschriebenen Control-Plane-Architektur. Sie sind gegenüber der v1.3 überarbeitet: Die Praxiserfahrung der Referenzimplementierung (Kapitel 19) und die Zielarchitektur der parallelen Agenten-Workflows (19.10) haben die ursprünglichen sechs Prinzipien geschärft und erweitert.

Prinzip	Kurzformel	Beschreibung
AP-1: Controlled Non-Determinism	Begrenzen statt vorhersagen	Das LLM-Ergebnis ist nicht deterministisch — kontrollierbar sind Zustände, Übergänge, Budgets, Policies, Schreibrechte, Freigaben, Gate-Regeln und Audit-Ereignisse. Die Architektur macht Agentenarbeit begrenzt und nachvollziehbar, nicht vorhersagbar.
AP-2: Single Writer per Workspace	Ein Schreiber je Arbeitskopie	Pro Branch, Worktree oder Workspace schreibt zu einem Zeitpunkt genau ein Agent. Das verhindert Schreibkonkurrenz, unklare Zurechnung und nicht attestierbare Zwischenstände.

Prinzip	Kurzformel	Beschreibung
AP-3: Parallelism by Dependency	Parallel nur, was unabhängig ist	Parallelität wird aus Task-Abhängigkeiten, Schreibbereichen, versionierten Verträgen und Risikopolicies abgeleitet — nicht aus festen Agentenrollen. Voneinander abhängige Komponenten werden erst parallel implementiert, wenn gemeinsame Verträge vorliegen.
AP-4: Independent Verification	Wer schreibt, gibt nicht frei	Erzeugung und Freigabe sind technisch getrennt: Read-only-Reviewer und Gates liefern strukturierte Befunde; ein Agent bewertet nie allein die eigene Arbeit.
AP-5: Policy as Executable Structure	Regeln als Code	Policies sind versionierter, signierter und attestierter Teil des Ausführungsmodells — keine Prosa-Dokumentation. Deklarative Repo-Konfiguration liefert den Kontext, technische Durchsetzung erfolgt über Tool-Rechte, Hooks und Gates.
AP-6: Human Authority	KI assistiert, Mensch entscheidet	Kritische Architektur-, Security-, Policy- und Deployment-Entscheidungen bleiben menschlich freigabepflichtig; die Freigabe ist ein Zustand im Prozess, kein Nebenkanal.

Prinzip	Kurzformel	Beschreibung
AP-7: Fail Closed	Ausfall ist kein Erfolg	Ein ausgefallener Pflicht-Reviewer, eine nicht prüfbare Policy oder eine fehlgeschlagene Attestierung darf niemals als Erfolg gelten. Ein Gate, dessen Ausfall wie Erfolg aussieht, ist schlimmer als kein Gate.
AP-8: Sovereign by Default	Läuft auch ohne Cloud	Die Architektur bleibt lokal, on-premises und ohne Cloud-Abhängigkeit betreibbar — bis hin zum Air-Gap-Betrieb. Skalierende Infrastruktur ist Option, nicht Voraussetzung.

Diese Prinzipien stehen nicht isoliert, sondern bedingen sich gegenseitig: Controlled Non-Determinism (AP-1) ist der Grund, warum es die übrigen sieben braucht. Single Writer (AP-2) macht Zurechnung und Attestierung erst möglich, auf denen Independent Verification (AP-4) und Policy as Executable Structure (AP-5) aufsetzen. Parallelism by Dependency (AP-3) definiert, wann AP-2 mehrfach nebeneinander existieren darf. Human Authority (AP-6) und Fail Closed (AP-7) sind die Sicherheitsnetze für alles, was AP-1 nicht einfangen kann. Und Sovereign by Default (AP-8) stellt sicher, dass das Ganze dort betreibbar bleibt, wo die Anforderungen am strengsten sind.

Was aus der Agentenspezialisierung wurde: Die v1.3 führte „Agent Specialization“ als oberstes Prinzip. Die Praxis hat das relativiert: Spezialisierung ist kein Architekturprinzip, sondern eine mögliche Ausführungs- und Routingstrategie — Rollen und Fähigkeitsprofile werden in den Kontext eines Laufs projiziert oder über Capability-Routing auf Modelle abgebildet (Kapitel 5, 19.4), ohne dass daraus mehrere gleichzeitig schreibende Prozesse folgen müssen.

Zuordnung zu Architekturkonzepten

Prinzip	Kapitelreferenzen	Durchsetzung
AP-1: Controlled Non-Determinism	Kap. 6 (Lifecycle), Kap. 7 (Execution Model)	Zustandsautomat, Turn-Limits, Execution Budgets, Timeouts
AP-2: Single Writer per Workspace	Kap. 16 (Workflows), Kap. 20 (ADR-2/ADR-5), 19.3/19.10	Branch-/Worktree-Isolation, Workspace Leases (seit 0.22.0, Feature-Flag)

Prinzip	Kapitelreferenzen	Durchsetzung
AP-3: Parallelism by Dependency	Kap. 16, 19.10	Task-Graph, Schreibbereiche, versionierte Verträge (seit 0.23.0, Feature-Flag)
AP-4: Independent Verification	Kap. 12 (Guardrails), 19.5	Read-only-Reviewer, Quality Gate, Claim Verification
AP-5: Policy as Executable Structure	Kap. 4 (Repo-Konfiguration), 19.6	Signierte Policy-Dokumente, Hooks, Gate-Policies
AP-6: Human Authority	Kap. 9 (Failure Handling), 19.3	Pflichtfreigaben als Prozesszustand, Segregation of Duties
AP-7: Fail Closed	Kap. 12, 19.5/19.7	Reviewer-Ausfall führt zu ERROR, Lizenz fail closed, Attestierung ohne Schlüssel führt zu Startfehler
AP-8: Sovereign by Default	Kap. 13 (Deployment), 19.7	Ein-Prozess-Betrieb, lokale Modelle, netzlose Sandbox, Lizenz ohne Rückkanal

Die Prinzipien und Referenzen wurden gegenüber v1.3 überarbeitet; die damaligen sechs Prinzipien (u. a. „Deterministic Execution“, das die Ausführung präziser als begrenzt und nachvollziehbar beschreibt) gehen in den neuen acht auf.

Praxis-Check SoftwareFabrik (erweitert): Die Prinzipien sind dort nicht beschrieben, sondern maschinell erzwungen: ArchUnit-Tests brechen den Build bei Schichten- oder Vendor-Kopplungs-Verstößen, und ein Debt-Ratchet friert bestehende Altschuld gezählt ein, statt sie wachsen zu lassen (19.2). AP-2 und AP-4 gelten bereits im Einzel-Run (ein schreibender Agent je Lauf, mehrere unabhängige read-only Reviewer). AP-3 ist vollständig implementiert (Feature-Flag, standardmäßig deaktiviert): Tasks werden aus einem Abhängigkeitsgraphen parallelisiert und laufen in getrennten Arbeitsverzeichnissen — zunächst nur lesend, inzwischen auch schreibend und gebunden an versionierte Verträge, deren Änderung betroffene Tasks automatisch auf Neuplanung setzt (Release-Verlauf: Tabelle in 19.10). AP-2 gilt dabei Workflow-weit: Ein Task startet nur, wenn seine Schreibbereiche mit keinem aktiven Task kollidieren. Auch der Regelkreis statt blindem Retry ist dort maschinell erzwungen — ein Task wird nur zurückgesetzt, wenn der Grund eine neue Eingabe trägt; Umsortieren allein genügt nicht (19.10).

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.

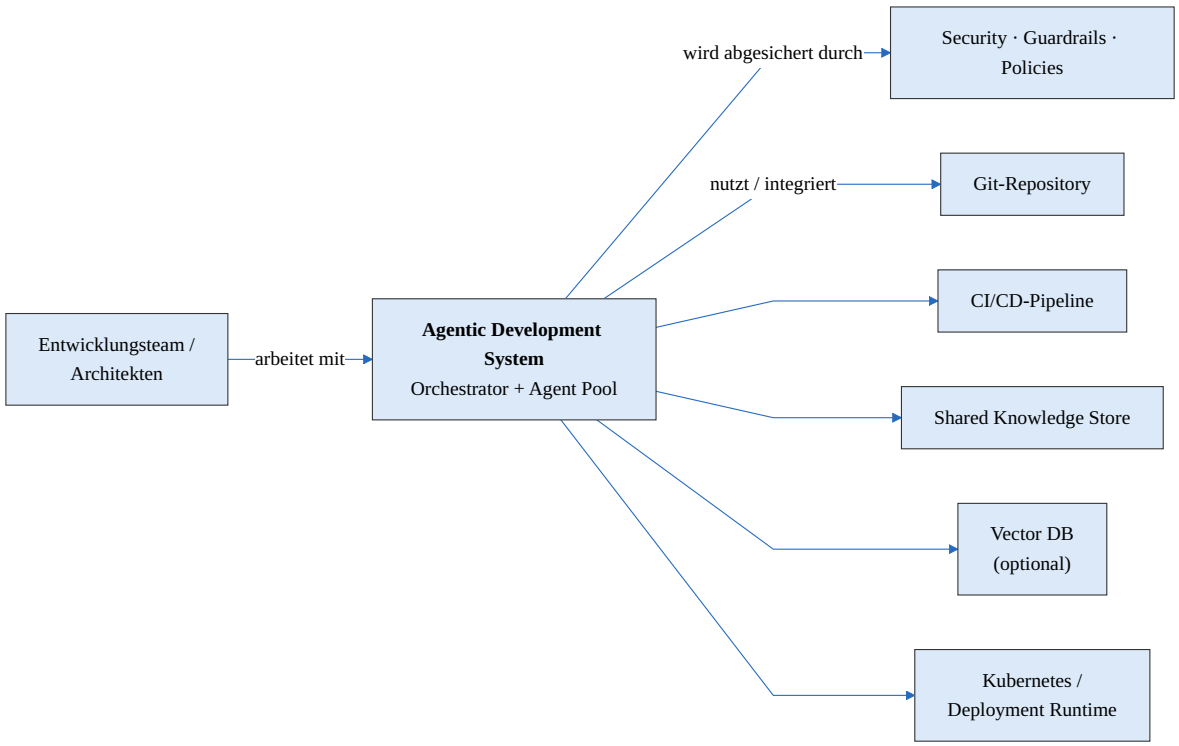

Abbildung 3.1.: Systemkontext des agentischen Entwicklungssystems

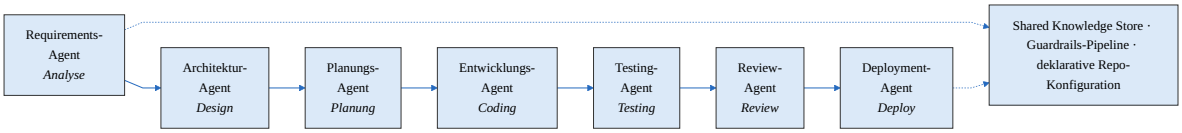

Abbildung 3.2.: SDLC-Agenten-Pipeline — von der Anforderungsanalyse bis zum Deployment

3.2. Referenzarchitektur des Agentensystems

Die Architektur folgt einem erweiterten Hub-and-Spoke-Modell: Im Zentrum steht der Orchestrator, der Arbeit an spezialisierte Rollen verteilt, deren Ergebnisse koordiniert und einen gemeinsamen Wissensstand (Shared Knowledge Store) verwaltet. Wichtig ist die Einordnung: Hub-and-Spoke ist eine **logische Rollen- und Kommunikationsstruktur** — ob Rollen als separate Agentenprozesse materialisiert werden, ist eine Implementierungsentscheidung. Die Fallstudie in Kapitel 19 zeigt beide Seiten: Kontextisolation verhindert Interferenzen, aber Parallelität erzeugt Konfliktkosten, wenn Schreibbereiche nicht abgegrenzt sind, und Spezialisierung ist nicht automatisch ein Qualitätsgewinn. Die aktuelle Referenzimplementierung verwendet deshalb einen schreibenden Agenten je Run und mehrere unabhängige Reviewer. Die Workflow-Ebene führt parallele Child Runs ein — seit Release 0.21.0 für nicht-schreibende Analyse, seit 0.22.0 auch schreibend, dort gebunden an abgegrenzte Schreibbereiche, erfüllte Abhängigkeiten und seit 0.23.0 an versionierte Verträge (AP-2/AP-3, Kapitel 19.10).

3.3. Agentenübersicht

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 begrenzte, nachvollziehbare 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) erzwingt die definierten Regeln für Änderungen, soweit sie technisch prüfbar sind; die verbleibenden Risiken behandelt Kapitel 12.

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

3.4. Runtime-Architektur

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.

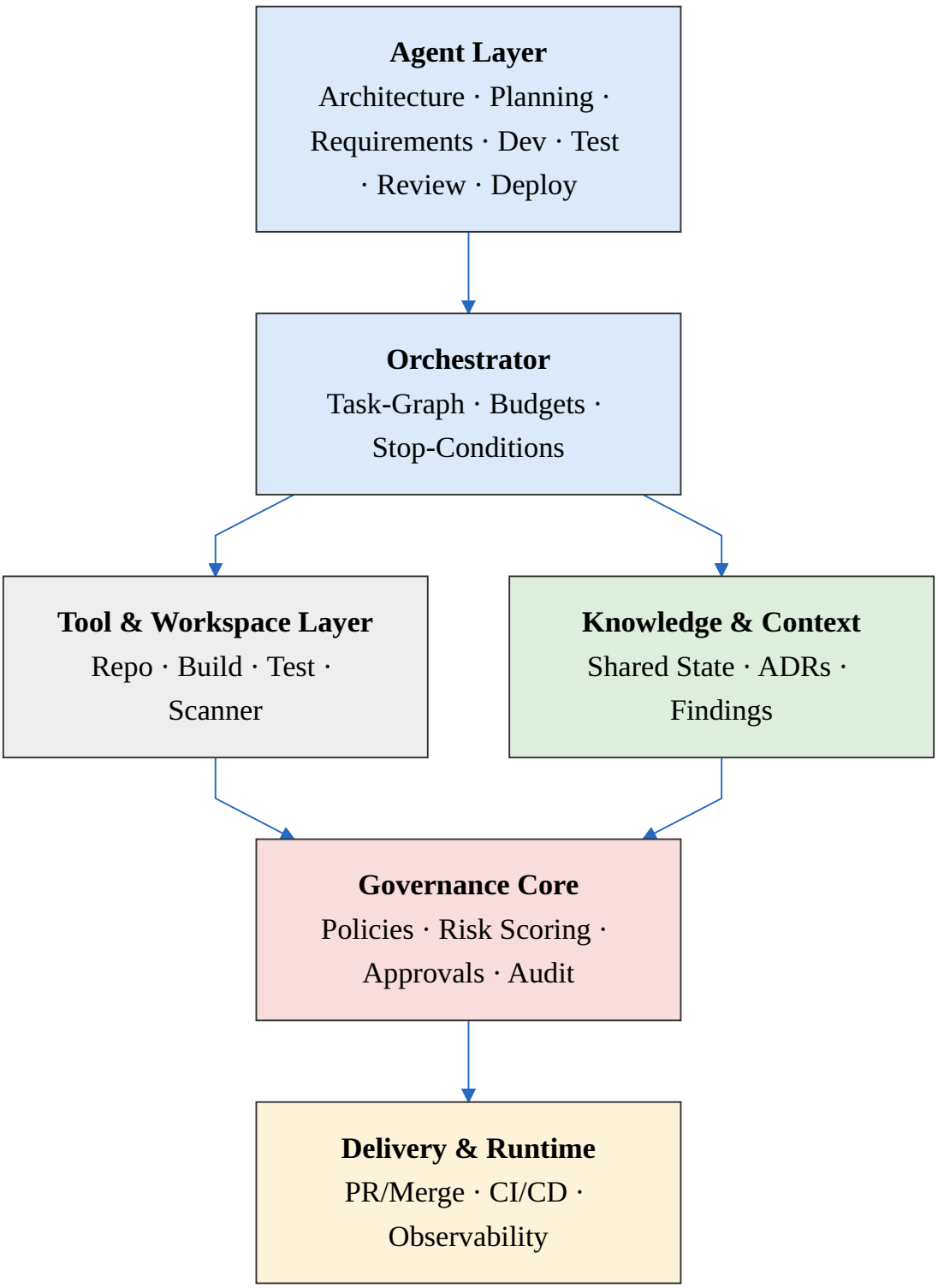

Abbildung 3.3.: Referenzarchitektur eines agentischen Entwicklungssystems im Enterprise-Kontext

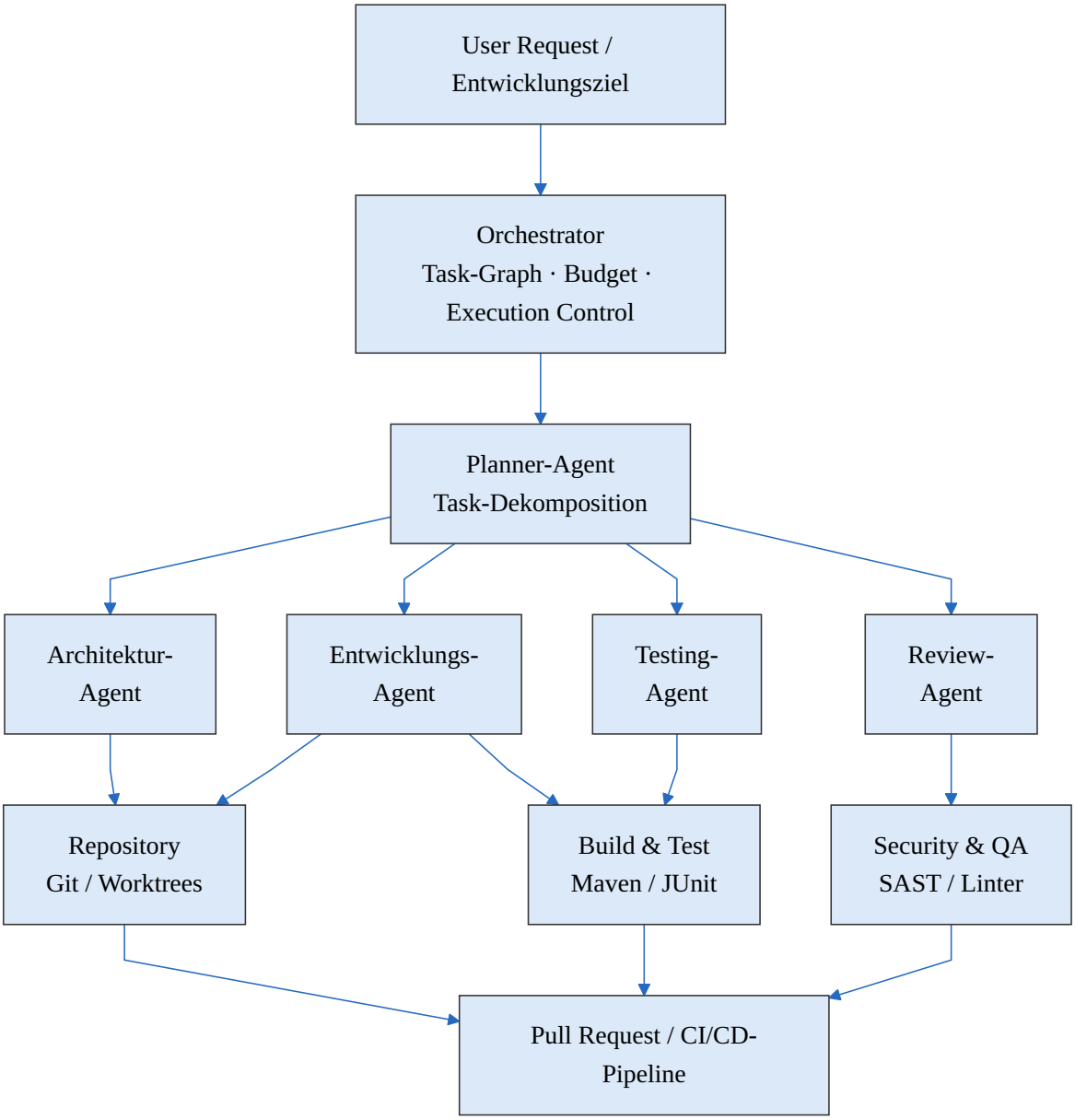

Abbildung 3.4.: Runtime-Architektur eines agentischen Entwicklungssystems

Agent	Rolle	Werkzeugzugriff
Orchestrator	Verteilt Aufgaben, koordiniert Phasen, verwaltet Shared State	Alle Tools inkl. Agent (Subagenten-Start)
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

Mehrere schreibende Rollen bedeuten dabei nicht gleichzeitiges Schreiben: Wie die Rollen auf tatsächliche Läufe abgebildet werden, regeln AP-2 und AP-3 (vgl. 3.2 und Kapitel 2).

3.5. Parameter des Agent-Tools (in v1.3: „Task-Tool“)

Das Werkzeug, mit dem der Orchestrator Subagenten startet, heißt in Claude Code inzwischen `Agent` (v1.3 dokumentierte es als „Task“). Die folgende Tabelle ist auf den Stand von August 2026 aktualisiert [4]; die Parameterliste ist schnelllebig und bei Umsetzung gegen die aktuelle Werkzeug-Dokumentation zu prüfen:

Parameter	Pflicht	Typ	Beschreibung
subagent_type	Ja	string	Welcher Agententyp gestartet wird (z. B. `code-reviewer`); eigene Typen werden als Markdown-Dateien mit Frontmatter unter `.claude/agents/` definiert
prompt	Ja	string	Detaillierte Aufgabenanweisungen mit Kontext und Erwartungen
description	Ja	string	Kurze Zusammenfassung der Aufgabe (laut Tool-Schema, für Anzeige und Logging)
model	Nein	enum	Modellklasse für diesen Agenten (sonnet, opus, haiku, . . .); ohne Angabe erbt der Agent das Modell der Hauptsitzung
run_in_background	Nein	boolean	Subagenten laufen inzwischen standardmäßig im Hintergrund; `false` erzwingt synchrone Ausführung

Die Worktree-Isolation (`isolation: worktree` — eine isolierte Git-Worktree-Kopie des Repositories je Agent) wird heute in der Subagenten-Definition gesetzt, nicht je Aufruf.

Gegenüber v1.3 bemerkenswert: Das damals als Aufrufparameter dokumentierte `max_turns` ist kein Parameter des Agent-Tools mehr — die Turn-Obergrenze wird heute als `maxTurns` in der Subagenten-Definition (Frontmatter) gesetzt. Und die Fortsetzung eines Agenten läuft nicht mehr über einen `resume`-Parameter, sondern über eine Nachrichtenschnittstelle an den — auch bereits beendeten — Agenten. Auch das illustriert eine Kernaussage dieses Whitepapers: Werkzeugdetails

haben eine Halbwertszeit von Monaten — Architekturprinzipien (ein Orchestrator, spezialisierte Subagenten mit eigenem Kontextfenster, Isolation) bleiben.

> **Praxis-Check SoftwareFabrik (abweichend):** Die Fabrik nutzt dieses Werkzeugmuster bewusst nicht: Sie startet keine Subagenten, sondern genau einen Agentenprozess je Lauf in einer Sandbox. Spezialisierung entsteht über Rollen- und Teamdefinitionen im Kontext und über mehrere unabhängige Prüfer nach der Ausführung (19.3, 19.8).

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. CLAUDE.md bzw. AGENTS.md stellt dem Agenten verbindlich formulierte Projektregeln als Kontext bereit — die deklarative Hälfte von AP-5 (Policy as Executable Structure). Technische Erzwingung entsteht erst durch Tool-Beschränkungen, Hooks, Sandbox und nachgelagerte Quality Gates.

> **Aktualisierung (Stand August 2026):** Seit v1.3 hat sich mit `AGENTS.md` ein werkzeugübergreifender De-facto-Standard für genau diese Datei etabliert — ein offenes Format, das inzwischen von den großen Coding-Agenten (u. a. Codex, GitHub Copilot, Cursor, Gemini CLI) unterstützt und unter dem Dach der Agentic AI Foundation der Linux Foundation gepflegt wird [1]. Claude Code liest weiterhin CLAUDE.md. Die Empfehlung für vendor-neutrale Projekte lautet daher: die Regeln **einmal** in `AGENTS.md` pflegen und werkzeugspezifische Dateien wie CLAUDE.md als minimale Verweise darauf anlegen — eine Quelle, mehrere Projektionen. Alles, was dieses Kapitel über Aufbau, Vererbung und Wirkung der Datei sagt, gilt unabhängig vom Dateinamen.

> **Praxis-Check SoftwareFabrik (erweitert):** Genau dieses Muster ist dort implementiert: Die Engineering-Guardrails liegen als eine versionierte Quelle vor und werden bei jedem Lauf als `AGENTS.md` plus minimaler CLAUDE.md-Verweis ins Repository projiziert; welche Version gewirkt hat, wird attestiert (19.6).

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

Geltungsbereich	Ort	Zweck
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 die Rollen-/Capability-Zuordnung (vgl. Kap. 2) 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-5)
Architekturregeln	Erlaubte Patterns, verbotene Anti-Patterns, Schichtentrennung	Review-Agent nutzt diese als Prüfkatalog
Cost Controls	Default-Modell, Budget-Limits, Eskalationsregeln	Steuert Execution Budget (Kap. 7) und Token Budget (Kap. 15)
Hooks & MCP	PreToolUse/PostToolUse Hooks, MCP-Server-Referenzen	Automatische Validierung als Teil von AP-5/AP-7

Sektion	Inhalt	Wirkung
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 produktionsnahe CLAUDE.md für ein fiktives Behördenprojekt der Exportkontrolle. Jede Sektion ist kommentiert, um die Wirkung auf das Agenten-Verhalten zu erklären:

```
# ============================================================
# Projekt: Behoerdenplattform Exportkontrolle (fiktives Beispiel)
# Team: Backend Engineering, Modernisierungsprogramm
# ============================================================

# --- Agentendefinitionen ---
# Jeder Agent hat: Name, Rolle, erlaubte Tools, Beschränkungen
# Dies setzt die Rollen-/Capability-Zuordnung (vgl. Kap. 2) 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. 15).

## 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, Claim Verification
- haiku NUR für: Code-Formatierung, einfache Umbenennung, Docs ohne Fachlogik

# --- Hooks ---
# Automatische Validierung bei jeder Tool-Nutzung (AP-5/AP-7).

## 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 – in der eigenen Projektpraxis dauert der Feedback-Loop erfahrungsgemäß 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 erfahrungsgemäß aus.

5. Eingebaute Agententypen und Modellauswahl

Claude Code bietet vorkonfigurierte Agententypen, die auf die häufigsten SDLC-Aufgaben zugeschnitten sind. Eigene Agententypen werden als Markdown-Dateien mit Frontmatter unter `.claude/agents/` definiert (v1.3 beschrieb hierfür noch CLAUDE.md; die zentrale Konfigurationsdatei steuert heute Regeln und Kontext, nicht die Agentendefinitionen) [4].

Agententypen und ihre SDLC-Phasen

Stand August 2026 liefert Claude Code als eingebaute Typen im Kern `Explore`, `Plan` und `general-purpose` [4]; die übrigen Einträge der folgenden v1.3-Tabelle sind als projektspezifisch definierbare Beispieltypen zu lesen:

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. Die Faustregel aus v1.3 gilt unverändert — nur die konkreten Modelle dahinter haben gewechselt: Die Klassenbezeichnungen `opus`, `sonnet` und `haiku` bleiben in Claude Code stabil, während die dahinterliegenden Modellversionen wechseln (Stand 6. August 2026: Claude Opus 5, Claude Sonnet 5, Claude Haiku 4.5; dazu ist mit `fable` — Claude Fable 5, dem leistungsfähigsten allgemein verfügbaren Modell für die schwierigsten Langzeit-Agentenaufgaben — seit v1.3 eine vierte Klasse hinzugekommen: Klassen sind also stabiler als Versionen, aber nicht unveränderlich) [11]. Also: die Opus-Klasse für Entscheidungen, bei denen Fehler teuer wären (Architektur, Security), die Sonnet-Klasse als Arbeitspferd für den Großteil der Entwicklung, und die Haiku-Klasse für schnelle, unkritische Aufgaben.

Die eigentliche Lehre aus der kurzen Halbwertszeit von Modellnamen: Ein System sollte **Fähigkeitsklassen** konfigurieren, nicht Modellnamen. Wo ein konkreter Modellname in Konfiguration oder Code steht, ist er in Monaten veraltet; eine Zuordnung „Rolle → Fähigkeitsstufe → aktuelles Modell“ bleibt stabil.

> **Praxis-Check SoftwareFabrik (erweitert):** Genau diese Abstraktion ist dort umgesetzt — Capability-Routing bildet Rollen (Planung vs. Umsetzung) auf Fähigkeitsprofile ab, die erst zur Laufzeit auf ein konkretes Modell aufgelöst werden; die Auflösung wird je Run attestiert (19.4).

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. 16) 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 Agent-Tool (in v1.3: „Task“, vgl. 3.5) mit Typ, Prompt, Modell und optionalen Parametern (`run_in_background`); die Turn-Obergrenze `maxTurns` und die Worktree-Isolation stehen in der Subagenten-Definition. 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.

#	Phase	Beschreibung
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 maxTurns-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).

#	Phase	Beschreibung
7	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. Er läuft vor dem endgültigen Übergang nach `TERMINATED` — der Endzustand erlaubt keine Übergänge mehr.
8	Termination	Der Agent beendet sich durch: (a) erfolgreichen Abschluss, (b) Erreichen von maxTurns, (c) Budget-Erschöpfung, (d) explizite Stop-Condition, oder (e) Fehler-Eskalation. Das Ergebnis wird an den Orchestrator zurückgegeben.

Praxis-Check SoftwareFabrik (erweitert): Der Lebenszyklus ist dort als *Run* modelliert: sieben Phasen, 13 Zustände, persistiert, pausier- und wiederaufnehmbar. Der Mensch ist als Zustand (`WAITING_FOR_APPROVAL`) Teil des Automaten, nicht Nebenprozess (19.3).

Versionshinweis (v2.2, gilt für Teil II): Die Java-Beispiele der Kapitel 6 bis 9 sind Modell-Skizzen zur Illustration des jeweiligen Konzepts, formuliert gegen Java 21 LTS und Spring Boot 3.x. Sie sind gekürzt (Imports, Konstruktoren, Exception-Handling teilweise weggelassen) und nicht als lauffähige Bibliothek getestet. Der Produktivstand des in Kapitel 19 beschriebenen Systems ist Java 25 und Spring Boot 4.0.

Java-Beispiel: Agent Lifecycle Manager

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

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

    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 nach eigener Betriebserfahrung (SoftwareFabrik, Stand August 2026) rund 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 16 (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-1 (Controlled Non-Determinism) und AP-5 (Policy as Executable Structure) in die Praxis um.

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, 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());
```

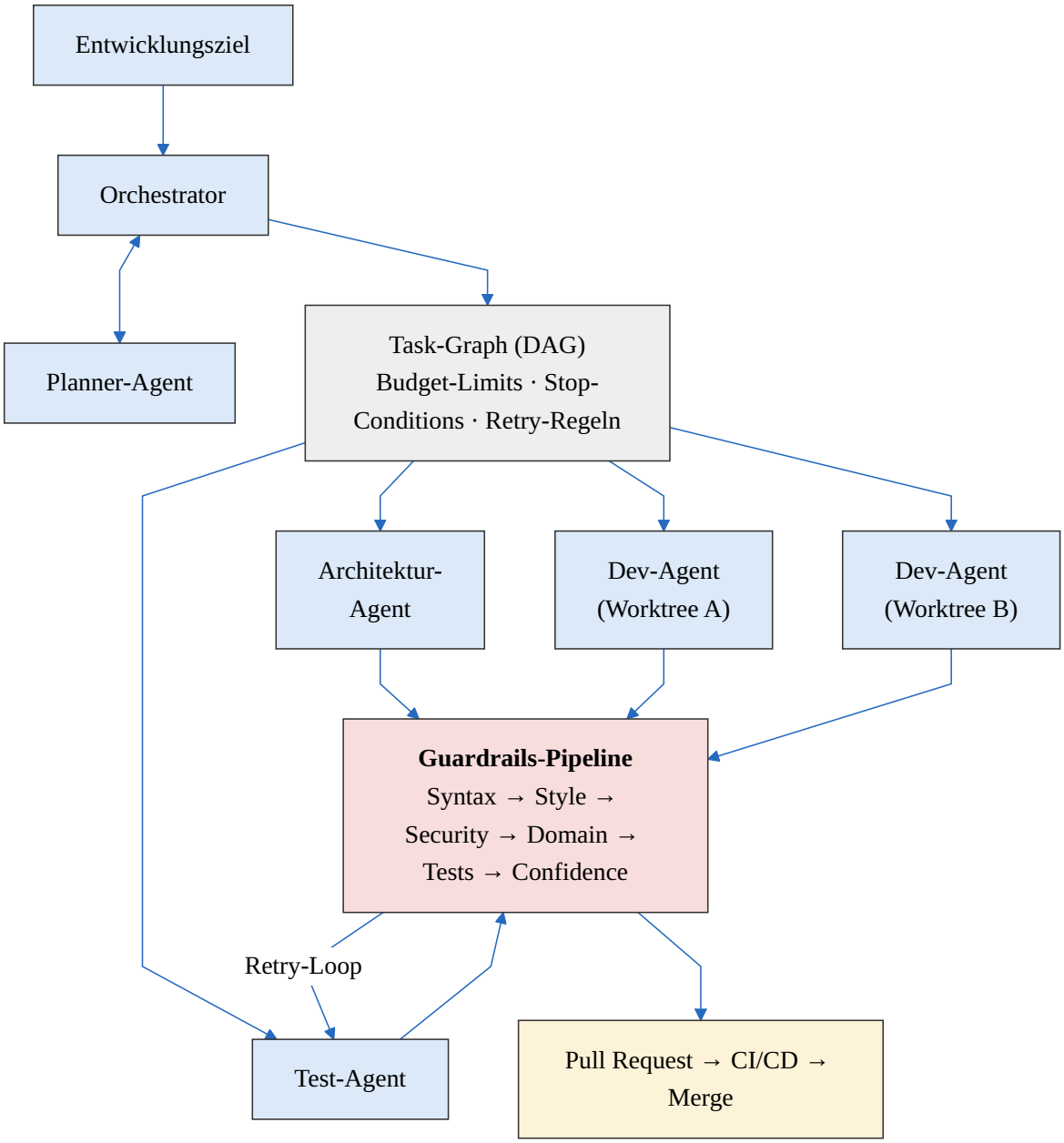

Abbildung 7.1.: Execution Pipeline — vom Entwicklungsziel über den Task-Graph bis zum Pull Request

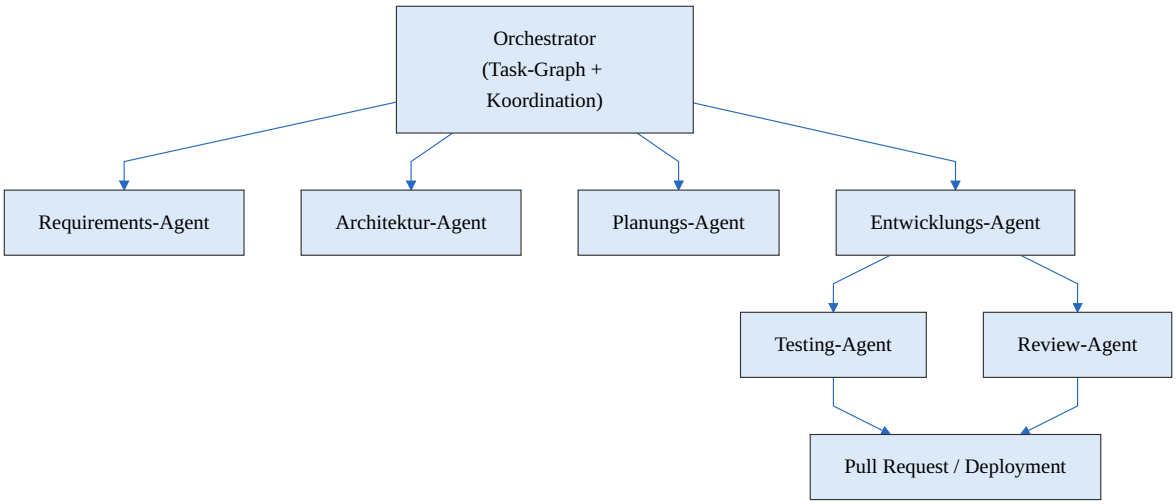

Abbildung 7.2.: Laufzeitablauf eines Entwicklungsauftrags

```
                    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.

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 — parallel jedoch nur für read-only-Schritte oder bei getrennten Workspaces mit abgegrenzten Schreibbereichen (AP-2/AP-3, Kapitel 2).

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

7.4. Execution Budget

Das Execution Budget ist ein zweilagiges Sicherheitsnetz: Es begrenzt sowohl einzelne Tasks (Turn-Limit (maxTurns), 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() {
        // maxParallelAgents gilt nur fuer read-only-Schritte oder Agenten in
        // getrennten Workspaces - ein Schreiber je Arbeitskopie (AP-2).
        return new ExecutionBudget(25, Duration.ofMinutes(10),
                new BigDecimal("10.00"), new BigDecimal("100.00"), 5, 7);
    }
}
```

7.5. 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

Praxis-Check SoftwareFabrik (abweichend, erweitert): Statt eines Task-Graphen je Auftrag eine lineare Phasen-Pipeline je Lauf — der eigentliche Graph liegt eine Ebene höher im Backlog mit Abhängigkeiten. Und aus der Retry-Strategie wurde ein Regelkreis: Build-Ausgabe, Reviewer-Findings, Merge-Konfliktdateien und CI-Status werden zur Eingabe des nächsten Laufs — ein Retry ohne neue Information wiederholt nur den Fehler (19.3, 19.8).

8. Memory Architecture

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:

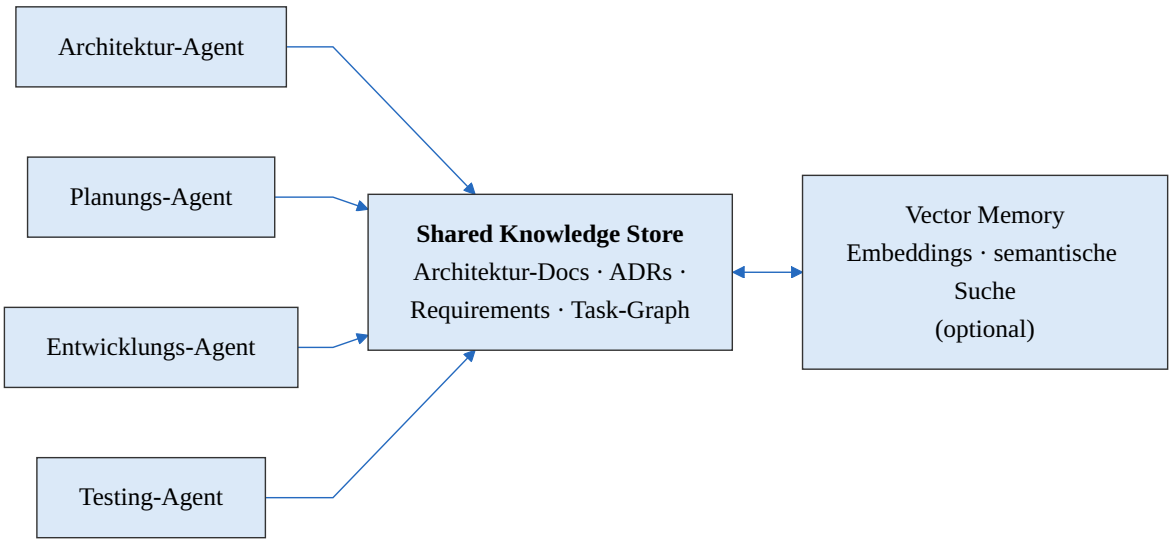

Abbildung 8.1.: Wissens- und Speicherarchitektur eines Multi-Agent-Entwicklungssystems

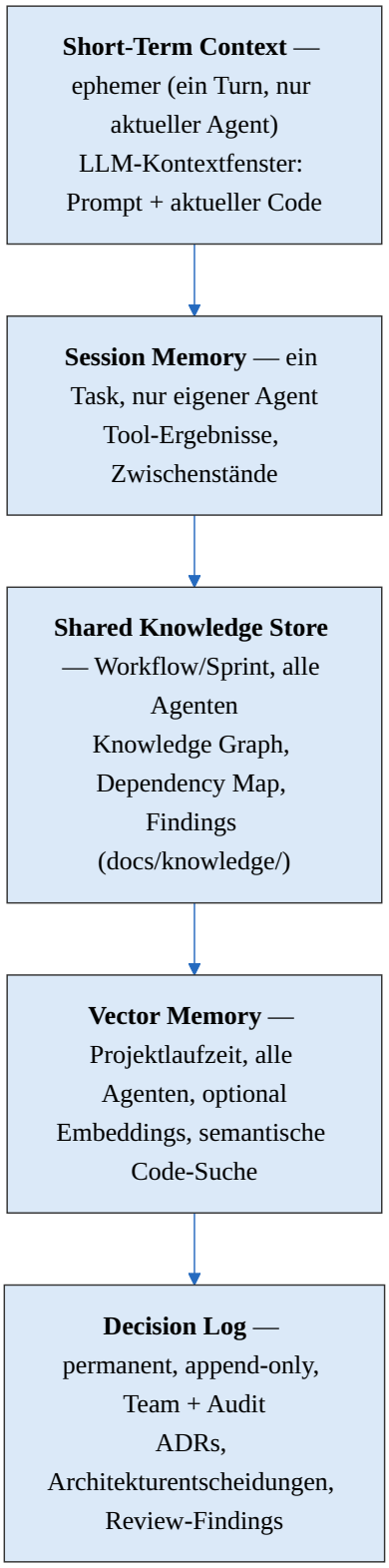

Abbildung 8.2.: Memory Architecture — Fünf-Schichten-Modell von ephemer bis persistent

Schicht	Lebensdauer	Sichtbarkeit	Implementierung
Short-Term Context	Ein Turn	Nur der aktuelle Agent	LLM Context Window (modellabhängig 200K–1M Tokens, Stand 08/2026)
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 — bei den aktuellen Claude-Modellen 1 Mio. Tokens, bei Haiku 4.5 200.000 Tokens (Stand 6. August 2026 [11]; v1.3 nannte hier noch 200K als Maximum — die Verfünffachung der Tokenzahl binnen weniger Monate (gemessen an der Textmenge wegen des neuen Tokenizers knapp das Vierfache) ist selbst ein Argument dafür, Kontextgrößen nie fest in Architekturentscheidungen einzubauen). 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:

Komponente	Beschreibung	Zugriffsmuster
Knowledge Graph	Entitäten und Beziehungen im Code	Write: architecture-agent | Read: alle
Context Cache	Kurzfristiger Sitzungs-Kontext	Write/Read: aktiver Agent

Komponente	Beschreibung	Zugriffsmuster
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 {
```

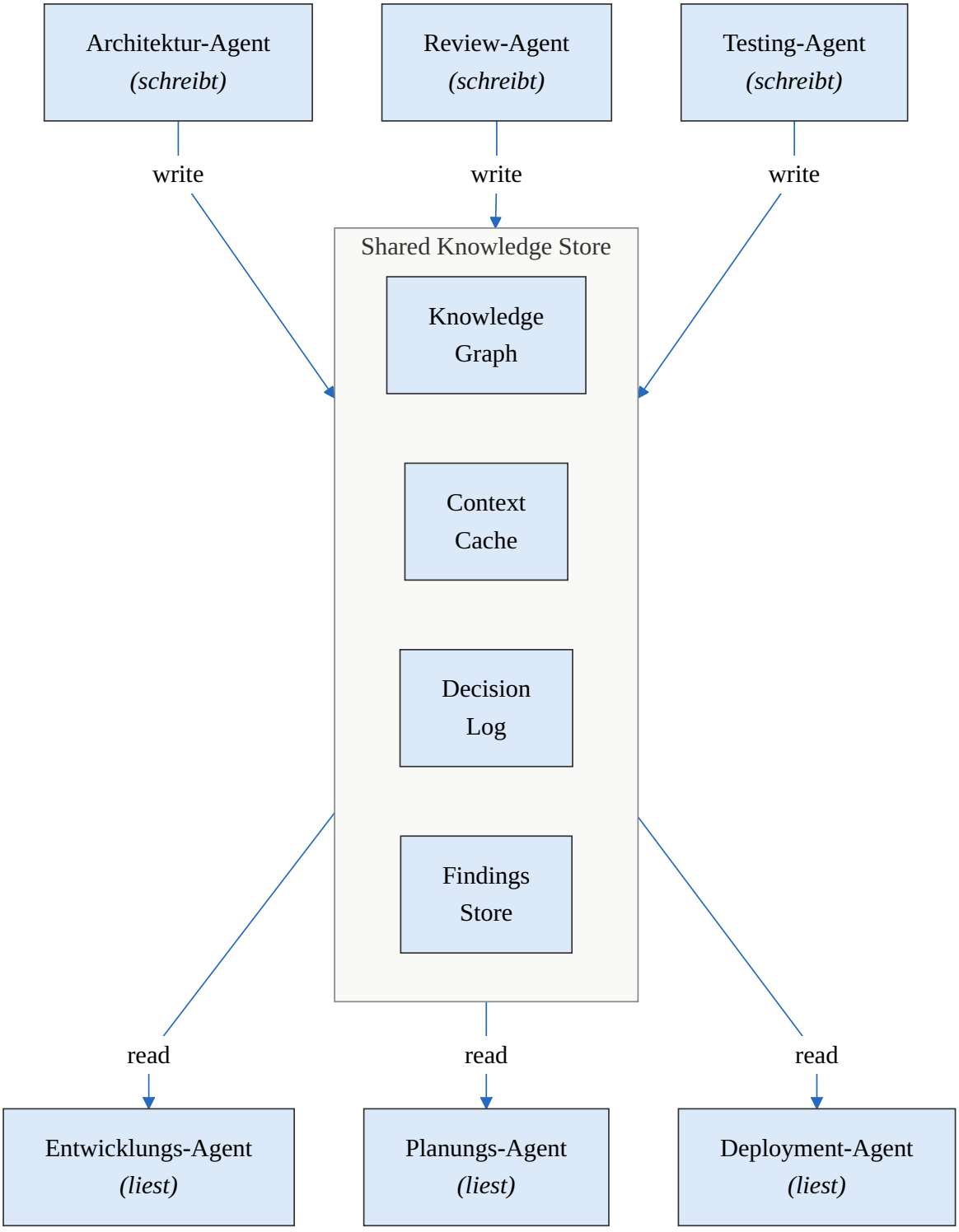

Abbildung 8.3.: Shared Knowledge Store — zentrale Wissensbasis für Agenten

```
            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`, ein offenes deutsch-englisches Modell von deepset und Mixedbread [5]). Dies ist insbesondere für Behörden-Projekte relevant, bei denen Cloud-basierte Embedding-Services aus Datenschutzgründen nicht in Frage kommen.

> **Praxis-Check SoftwareFabrik (abweichend):** Der geteilte Wissensstand ist umgesetzt — als versionierte Dateien im Repository plus kuratiertes Projektgedächtnis. Auf den optionalen Vektorspeicher wurde bewusst verzichtet: Was der Agent liest, muss ein Mensch reviewen und ein Auditor zitieren können (19.6).

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 Claim Verification).	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

#	Strategie	Beschreibung	Anwendungsfall
5	Compensation	Gegenläufige Aktion zur Wiederherstellung eines konsistenten Gesamtzustands bei verteilten Workflows.	Wenn parallele Worktrees inkonsistent werden

Java-Beispiel: Failure Handler mit Eskalationslogik

```java
// === 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 }
}
```

Praxis-Check SoftwareFabrik (erweitert): Fehlschlag ist dort ein regulärer Zustand mit definiertem Ausgang (Korrekturschleife, Terminalzustände). Zentrale Regel über das Konzept hinaus: Der Ausfall eines Prüfmechanismus darf nie wie Erfolg aussehen — ein abgestürzter Reviewer führt zu `ERROR`, die Lizenzprüfung ist *fail closed* (19.5, 19.7).

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 Agenten-Aufrufen und Java-Enterprise-Codebeispielen vorgestellt.

Die sieben Agenten sind als logische Rollen beziehungsweise Capabilities zu verstehen. Sie müssen nicht als sieben separate Prozesse materialisiert werden und arbeiten insbesondere nicht zwangsläufig gleichzeitig schreibend am selben Vorhaben. Wie die Rollen auf tatsächliche Ausführungen abgebildet werden — auf einen einzelnen Run mit Capability Routing oder auf mehrere isolierte Child Runs —, entscheidet die Prozessschicht (Kapitel 2, 19.4, 22).

> **Versionshinweis (v2.2):** Die Java-Codebeispiele dieses Teils stammen aus v1.3 und sind gegen Java 21 LTS und Spring Boot 3.x formuliert; sie wurden bewusst nicht auf neuere Versionen gehoben und nicht erneut dagegen getestet. Es sind **gekürzte Auszüge**: Imports, Konstruktor-Annotationen und Exception-Handling sind teilweise weggelassen, damit die Beispiele auf die jeweils gezeigte Struktur fokussieren — sie sind in dieser Form nicht isoliert übersetzbar. Für den Sprung auf Spring Boot 4 (GA seit November 2025) siehe die dokumentierten Fallstricke des Realsystems in Kapitel 19 — das dort beschriebene System läuft auf Java 25 und Spring Boot 4.0 (u. a. Wechsel auf Jackson 3 als primären Serialisierer). Aufruf-Skizzen (`Task(...)`) sind Pseudocode auf v1.3-Werkzeugstand; das Werkzeug heißt heute `Agent` (vgl. 3.5).

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-2, 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 9457, vormals 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

```yaml
# 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 }
```

Praxis-Check SoftwareFabrik (abweichend): Die sieben Rollen existieren dort teils als Agentenrollen im Kontext (Definitionen, Teams), teils als Systemfunktionen: Planung = Plan-Run, Review = Read-only-Reviewer-Schicht, Testing/Deployment = Build-Gate und Meilenstein-Release. Die Parallelität wurde von der Erzeugung auf die Bewertung verschoben — ein Agent je Lauf, mehrere Prüfer (19.3, 19.5, 19.8).

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.

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

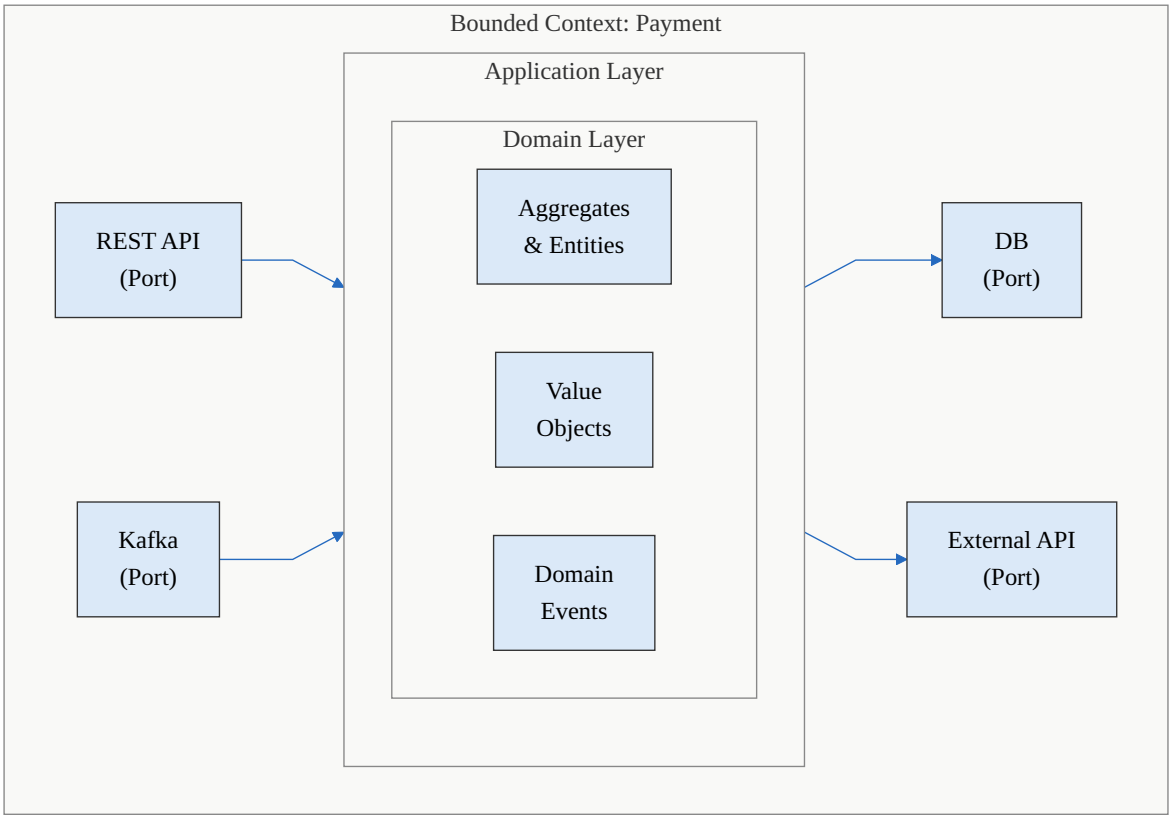

Abbildung 11.1.: Hexagonale Architektur mit DDD und Agenten-Zuordnung

DDD-Konzept	Agenten-Integration	Beispiel
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()); }
        });
    }
}
```

Praxis-Check SoftwareFabrik (abweichend): Die Fabrik wendet DDD auf sich selbst an — modelliert wird der Entwicklungsprozess als Domäne (27 Bounded Contexts: Run, Backlog, Freigabe, Policy, Nachweis). Bewusste Pragmatik gegen die reine Lehre: Aggregat und JPA-Entität sind dieselbe Klasse; statt Kafka genügen interne Events im Einzelprozess-Deployment (19.1, 19.2).

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.

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.

Die Pipeline ist als Defense-in-Depth-Modell angelegt: Mehrere unterschiedlich arbeitende Prüfungen reduzieren das Risiko, dass ein einzelner Fehler unbemerkt bleibt. Die ersten fünf Stufen prüfen überwiegend deterministische Kriterien; die LLM-basierte letzte Stufe kann zusätzliche kontextuelle Auffälligkeiten identifizieren, die vom konfigurierten Regelwerk nicht erfasst werden — sie ist damit selbst nichtdeterministisch. Der Prüf**prozess** ist reproduzierbar, die Prüf**ergebnisse** sind es auf dieser Stufe nur eingeschränkt. Eine vollständige Fehlererkennung wird nicht behauptet.

Guardrails erzwingen definierte technische Mindestkriterien, liefern strukturierte Befunde und reduzieren das Risiko ungeprüfter Änderungen. Sie ersetzen weder fachliche Abnahme noch unabhängige Sicherheitsprüfung oder Produktionsbeobachtung.

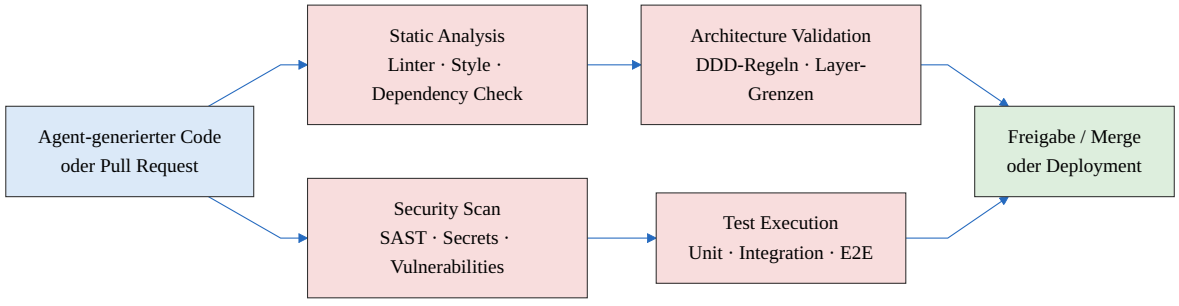

Abbildung 12.1.: Validierungs- und Governance-Pipeline

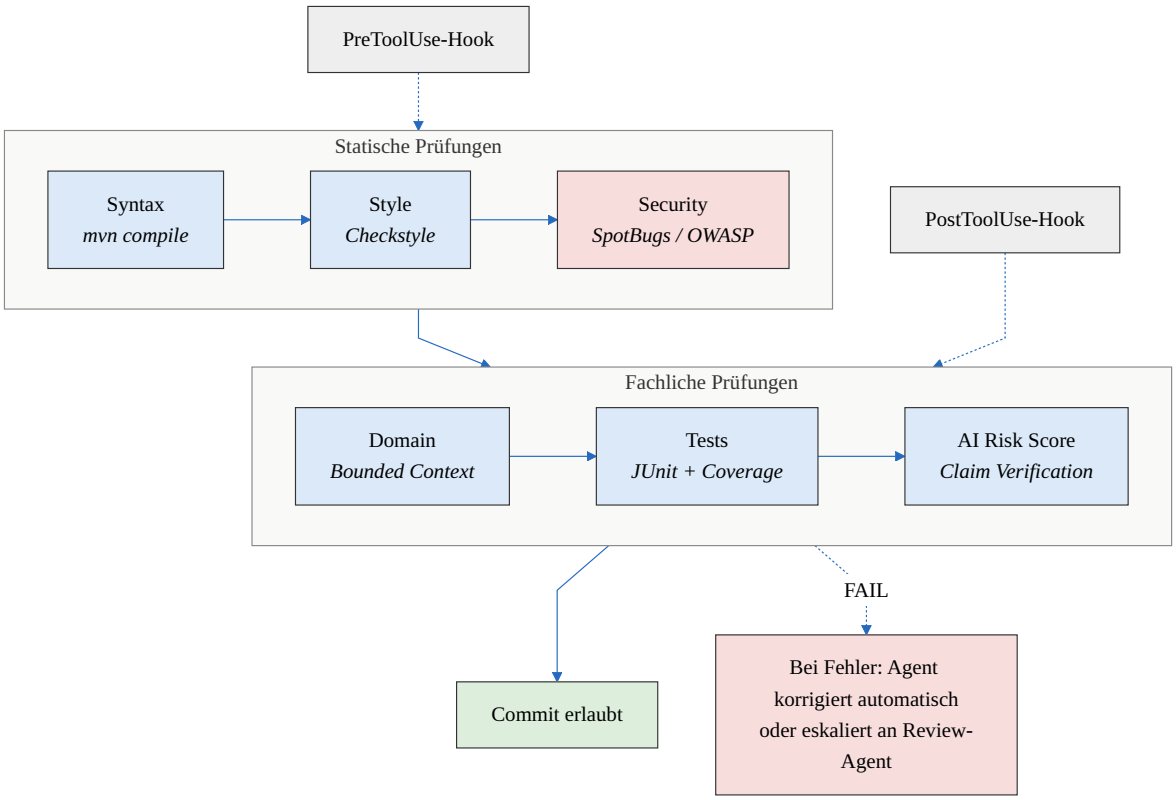

Abbildung 12.2.: AI-Guardrails-Pipeline zur Validierung agentisch erzeugter Codeänderungen

Komponente	Funktion	Umsetzung
Claim Verification (in v1.3: 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})
```

```java
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

Praxis-Check SoftwareFabrik (erweitert): Die Guardrails-Pipeline ist als eigener Schichtbegriff umgesetzt: Read-only-Review-Adapter, getrennt von den schreibenden Agenten. Sechs Review-Adapter — nicht deckungsgleich mit den sechs Pipeline-Stufen oben: zwei LLM-Reviewer, drei statische Prüfer, ein werkzeuggestützter Dependency-Scan —, eine konfigurierbare Gate-Policy mit drei nicht aufweichbaren Sonderregeln

und drei Betriebsmodi (off/advisory/blocking); das Confidence Scoring ist ohne AOP im Gate-Service gelöst (19.5).

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.

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

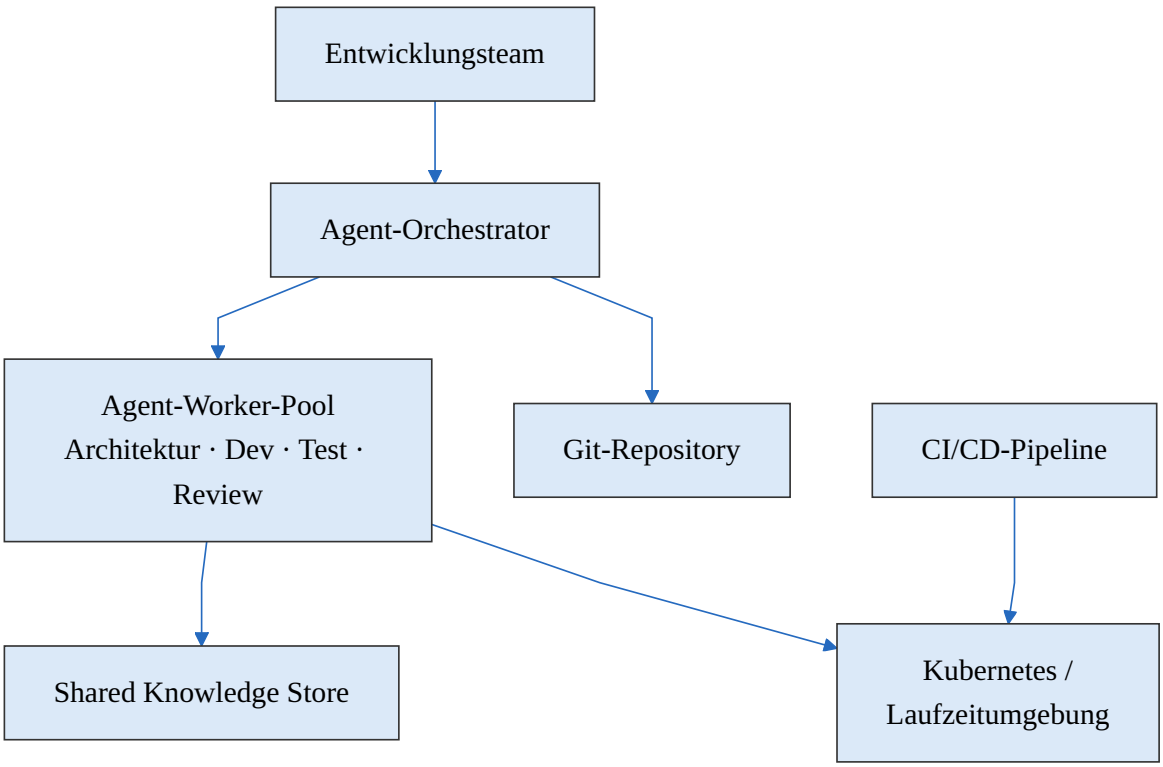

Abbildung 13.1.: Deployment-Architektur eines agentischen Entwicklungssystems

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.

Einordnung *(neu in v2.0)*

Die hier gezeichnete Kubernetes-Architektur ist als **Option** zu lesen, nicht als Voraussetzung. Als Erfahrungswert aus der Arbeit an der in Kapitel 19 beschriebenen Plattform: Reale Zielgruppen im regulierten Umfeld — Behörden, Finanzdienstleister, Mittelstand — stellen vor der Skalierungsfrage eine andere: *Läuft es bei uns, on-premises, ohne Cloud-Abhängigkeit, notfalls ohne Rückkanal?* Ein agentisches Entwicklungssystem kann als einzelner Prozess mit einer Datenbank betrieben werden; die eigentliche Komplexität liegt in der Steuerung des Nichtdeterminismus, nicht in der Betriebsinfrastruktur. Kubernetes bleibt der richtige Weg, wo elastische Skalierung vieler paralleler Agenten gebraucht wird — es ist aber nicht die Eintrittskarte.

> **Praxis-Check SoftwareFabrik (abweichend):** Ein Anwendungscontainer, eine Datenbank, Docker Compose; ephemere Agent-Container optional. Air-Gap-fähig bis hin zur Lizenz ohne Online-Aktivierung — Souveränität war das Entscheidungskriterium, nicht Skalierung (19.7).

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-4)
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 generiertem 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-2), 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
        // High-entropy hex (SHA1-like). Achtung: matcht auch jede
        // Git-Commit-SHA - in der Praxis False-Positive-Filter nötig
        // (z. B. Allowlist für .git-Pfade und Lockfiles)
        Pattern.compile("[0-9a-f]{40}")
    );

    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, Math.min(20,
 pat.pattern().length())) + "...",
                                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);
    }
}
```

Praxis-Check SoftwareFabrik (erweitert): Fünfstufiges RBAC, Mandantenisolation mit festgeschriebenem IDOR-Test, Umgebungs-Allowlist, Container-Sandbox mit `--network=none`, Host-Allowlist gegen Token-Exfiltration. Die härtesten realen Befunde kamen aus adversarialen Re-Reviews und lagen an den Übergängen zwischen Automatismen — nicht in den Funktionen selbst (19.4, 19.6, 19.9).

15. Wirtschaftlichkeitshypothese, Kostenmodell und Messplan

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

Preisstand 6. August 2026, API-Listenpreise von Anthropic [11, 12] (je 1 Mio. Tokens; Kontextfenster 1 Mio. Tokens, Ausnahme Haiku 4.5 mit 200.000):

Modell	Input (pro 1M Tokens)	Output (pro 1M Tokens)	Typischer Einsatz
Claude Fable 5	\$10.00	\$50.00	schwierigste Langzeit-Agentenaufgaben
Claude Opus 5	\$5.00	\$25.00	Architektur, Security Review, komplexe agentische Entwicklung
Claude Sonnet 5	\$2.00 (Einführungspreis bis 31.08.2026; danach \$3.00)	\$10.00 (danach \$15.00)	Entwicklung, Testing, Planung
Claude Haiku 4.5	\$1.00	\$5.00	Formatierung, einfache Tasks

Bemerkenswert ist der Vergleich mit der v1.3-Tabelle: Sie nannte für die Opus-Klasse noch \$15/\$75 — die Preise der Opus-4/4.1-Generation. Schon Opus 4.5 (November 2025, also vor Erscheinen der v1.3) lag bei \$5/\$25; die Tabelle war bei Drucklegung bereits überholt. Die Bewegung ist dabei nicht monoton nach unten: Die Haiku-Klasse wurde über zwei Generationen hinweg viermal teurer je Token — von Haiku 3 (\$0.25/\$1.25) über Haiku 3.5 (\$0.80/\$4) auf Haiku 4.5 (\$1/\$5) —, und seit der Opus-4.7-Generation erzeugt ein neuer Tokenizer für denselben Text bis zu rund 35 % mehr Tokens (inhaltsabhängig; Anthropics Migrationsleitfaden nennt einen Faktor von 1,0x bis 1,35x, der Modellüberblick „rund 30 %“) [9, 11] — Token-Preise sind generationsübergreifend also nur mit Vorbehalt vergleichbar. Die Lehre: Preistabellen in Kostenmodellen für

agentische Entwicklung brauchen zwingend ein Stand-Datum, und Wirtschaftlichkeitsrechnungen sind in beide Richtungen schnell veraltet.

Als Modellannahme dieses Kapitels — Erfahrungswerte, nicht systematisch gemessen (vgl. den Messplan in 15.6): 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
maxTurns (in der Subagenten-Definition; v1.3: max_turns als Aufrufparameter)	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

Die Dollarwerte dieser Tabelle sind eine Modellrechnung auf v1.3-Preisstand (März 2026); vgl. die aktuelle Preistabelle in 15.1.

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!)

In dieser vereinfachten Modellrechnung bleiben die reinen Ausführungskosten unveränderter Einzeltasks unabhängig von ihrer zeitlichen Anordnung gleich. **Nicht berücksichtigt sind** zusätzlicher Kontextaufbau je paralleler Ausführung, Planungs-, Synthese- und Integrationsschritte, Merge-Konflikte und Revalidierung — die Gesamtkosten eines parallelen Workflows sind daher nicht notwendigerweise identisch, sondern in der Regel höher. Besonders teuer wird es bei Merge-Konflikten und Retry-Schleifen. Deshalb ist Worktree-Isolation (AP-2) bei paralleler Ausführung Pflicht.

15.4. ROI-Berechnung

> **Hinweis:** Die folgende Gegenüberstellung ist eine Modellrechnung auf Basis der genannten Annahmen (Stundensätze, Aufwände, Token-Verbrauch) — keine gemessene Betriebsauswertung. Eine kontrollierte Messung liegt auch für das in Kapitel 19 beschriebene Realsystem nicht vor (siehe 19.9).

Kostenfaktor	Manuell (Entwicklerteam)	KI-Agenten + Review
Entwickleraufwand	40h Senior Dev × €95/h = €3.800	8h Review + Steering = €760
API-Kosten	–	~2M Tokens ≈ €15–30
Testabdeckung (angenommen)	Oft <60% unter Zeitdruck	>80% durch iteratives Testing
Time-to-Feature (angenommen)	1–2 Wochen	1–2 Tage
Gesamtkosten	€3.800+	€775–€790

Der ROI hängt stark von der Aufgabenkomplexität ab: Bei Standard-CRUD-Features wird der Hebel am größten angenommen (modellhaft 5–10x; nicht gemessen). Bei komplexen Architekturentscheidungen sinkt der Automatisierungsgrad, aber der Analyse-Output (ADRs, Findings) kann die menschliche Entscheidungsfindung beschleunigen (Erfahrungswert, nicht gemessen; vgl. 15.6).

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, Claim Verification
- haiku für: Formatierung, Linting, Docs ohne Fachlogik
- sonnet für: alles andere (Default)
```

15.5. Abo- statt Token-Abrechnung *(neu in v2.0)*

Das bisherige Kapitel rechnet ausschließlich in Token-Preisen. Die Realität agentischer Entwicklung hat sich seit v1.3 an einer entscheidenden Stelle verschoben: Die großen Coding-CLIs lassen sich auf **zwei Wegen** authentifizieren — per API-Key (Abrechnung je Token, wie oben modelliert) oder per **Abo-Login** (Flatrate-Konto des Anbieters, etwa die Claude-Abonnements für Claude Code [2] oder das ChatGPT-Konto für die Codex-CLI [3]; Stand August 2026).

Für das Kostenmodell hat das grundlegende Konsequenzen:

1. **Die Grenzkosten je Lauf sind im Abo-Modus bis zur Limitgrenze praktisch null.** Ein reines Token-ROI-Modell (wie in 15.4) bildet die Wirtschaftlichkeit dann nicht mehr ab; an die Stelle variabler API-Kosten tritt ein fixer Abo-Preis je Entwicklerplatz und Monat, gedeckelt durch die Nutzungslimits des jeweiligen Abos.
2. **Die Betriebsform wird zur Kostenentscheidung.** Einzelplatz mit Abo, Team-Pool mit API-Keys oder Mischformen unterscheiden sich in Planbarkeit (fix vs. variabel), Attribution (je Platz vs. je Verbrauch) und Skalierungsverhalten.
3. **Der Abrechnungsweg muss technisch kontrolliert werden.** Eine CLI, die sowohl einen Abo-Login als auch einen API-Key in der Umgebung vorfindet, wählt unter Umständen stillschweigend den kostenpflichtigen Pfad — bei Claude Code etwa hat ein gesetzter `ANTHROPIC_API_KEY` Vorrang vor dem Abo-Login [2] —, ein Fehler, der erst auf der Monatsrechnung sichtbar wird. Wer beide Wege betreibt, braucht eine Stelle im System, die den jeweils nicht gewollten Weg aktiv unterbindet.

Diese Einordnung ist ein Erfahrungswert aus dem Betrieb der in Kapitel 19 beschriebenen Plattform; die dortige Umsetzung (Abo-Modus je Adapter mit aktivem Entfernen des API-Keys aus der Prozessumgebung) steht in 19.4.

Praxis-Check SoftwareFabrik (erweitert): Preistabelle je Modell mit Input-, Output- und Cached-Input-Preisen, Kostenaggregation nach Projekt, Run, Provider, Mandant und Seat, harte Budget-Caps je Mandant — und der Abo-Modus als eigener Authentifizierungsweg, der den API-Key aktiv aus der Subprozess-Umgebung entfernt (19.4). Die ROI-Modellrechnung aus 15.4 bleibt dagegen unbelegt: Für das Realsystem existiert keine kontrollierte Produktivitätsmessung, und eine Budget-Obergrenze je Nutzer ist offen (19.9).

15.6. Messplan *(neu in v2.1)*

Die Modellrechnungen dieses Kapitels werden erst durch Messung zu Ergebnissen. Für die in 19.10 beschriebene Weiterentwicklung ist deshalb begleitend ein Messprogramm vorgesehen, das drei Vorgehensweisen vergleicht — manuelle Umsetzung, Einzel-Run und paralleler Workflow — mit folgenden Messgrößen:

- menschliche aktive Arbeitszeit je Vorhaben
- Time to Accepted Merge
- First-Pass-Gate-Rate (Anteil der Läufe ohne Korrekturschleife)
- Korrekturschleifen und Replans
- Kosten je Child Run, je Workflow und je akzeptierter Änderung
- Merge-Konfliktrate
- entkommene Defekte und Rollbacks
- Reviewzeit und Testabdeckungsänderung

Praxis-Check SoftwareFabrik (Stand 0.30.0): Ein Teil dieses Messplans ist seit Release 0.27.0 implementiert (19.10, Stufe 6): Time to Accepted Merge, Erstdurchlauf-Quote, Korrekturschleifen, Planänderungen, Kosten je Workflow und Child Run, Merge-Konfliktquote und Freigabe-Wartezeit werden aus ohnehin entstehenden Daten abgeleitet; seit 0.28.0 kommt die Testabdeckungsänderung hinzu (in Prozentpunkten, erst ab der zweiten Messung), seit 0.29.0 die Rollback-Quote aus der Git-Historie (als Untergrenze ausgewiesen), seit 0.30.0 die Eingriffe und die Entscheiderzahl aus der Freigabeentscheidung — dazu ein erfragter, freiwilliger Aufwand, der nur zusammen mit seiner Abdeckungsquote ausgewiesen und nie mit Gemessenem verrechnet wird. Drei Messgrößen weist die Plattform bewusst als Lücke aus, statt sie zu schätzen: die menschliche aktive Arbeitszeit (Wartezeit ist nicht Arbeitszeit; seit 0.30.0 in messbare Eingriffe und erfragten Aufwand zerlegt — die Arbeitszeit selbst bleibt Lücke), entkommene Defekte (sauber messbar erst über eine Ticketsystem-Anbindung, die es nicht gibt; die Rollback-Hälfte dieser Lücke ist seit 0.29.0 geschlossen) und der Vergleich zur manuellen Umsetzung, dessen Referenzgruppe im System nicht existiert. Gemessen wird überwiegend auf der Workflow-Ebene, die weiterhin hinter dem standardmäßig deaktivierten Feature-Flag steht; allein der Einzel-Run-Teil der Rollback-Erkennung und das Aufwandsfeld an der Run-Freigabe wirken auch ohne Flag. Der dreiarmige Vergleich dieses Messplans steht damit weiter aus.

Bis diese Messungen vorliegen, sind alle Produktivitäts- und ROI-Aussagen dieses Kapitels als Hypothesen bzw. Modellrechnungen zu lesen (vgl. 15.4, 19.9).

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.

Einordnung (v2.1): Dieses Kapitel beschreibt ein Werkzeugmuster — die Subagenten-Aufrufe eines Coding-Agenten —, nicht den heutigen Ablauf der Referenzimplementierung. Die dort gezeigte Parallelität betrifft Analyse und Review, also lesende Arbeit; gleichzeitig **schreibende** Agenten setzen die Workspace-Isolation und Koordination der Zielarchitektur voraus (19.10, 22.4). Der Praxis-Check am Kapitelende ordnet das im Detail ein.

Die Aufruf-Skizzen sind auf den Werkzeugstand von August 2026 aktualisiert (Agent-Tool statt „Task“, vgl. 3.5) und als Pseudocode zu lesen [4]:

Sequenzieller Workflow

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

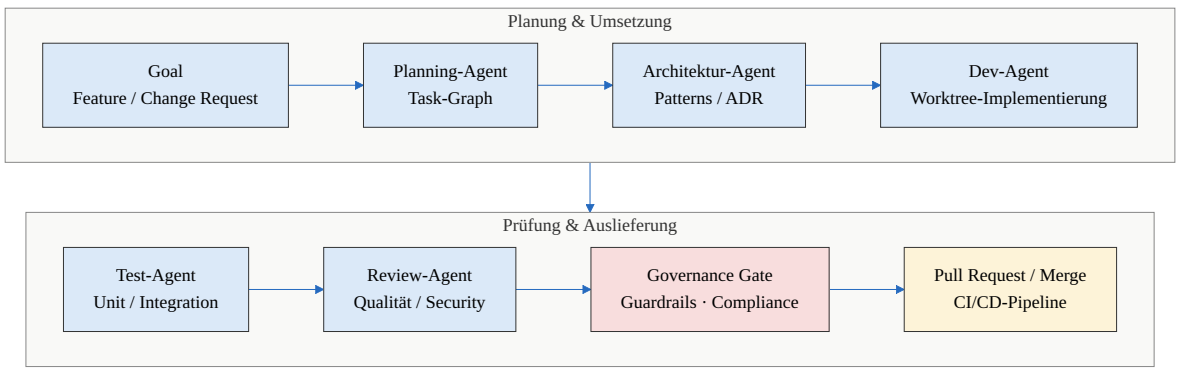

Abbildung 16.1.: Sequenzieller Multi-Agent-Workflow im Software Development Lifecycle

Paralleler Workflow

```
// Diese drei Agenten laufen PARALLEL (heute der Standardfall:
// Subagenten starten im Hintergrund). Alle drei sind LESEND - parallel
// schreibende Agenten brauchen getrennte Workspaces (AP-2, vgl. 3.5:
// isolation: worktree):
Agent(subagent_type="test-gap-agent",   prompt="Analysiere Testlücken...")
Agent(subagent_type="arch-agent",       prompt="Architektur-Review...")
Agent(subagent_type="review-agent",     prompt="Security-Review...")
```

Background & Wiederaufnahme

```
Agent(subagent_type="test-agent", prompt="mvn test")   // läuft im Hintergrund
// Später fortsetzen: Nachricht an den (auch bereits beendeten) Agenten
// (v1.3: eigener resume-Parameter; heute Nachrichtenschnittstelle)
SendMessage(to="agent_abc123", message="Korrigiere die Fehler.")
```

Praxis-Check SoftwareFabrik (abweichend): Sequenzielle, wiederaufnehmbare Abläufe sind umgesetzt; die parallele Multi-Branch-Ausführung mehrerer Läufe je Projekt ist bewusst zurückgestellt — ein Workspace je Projekt, damit Folgeläufe auf dem Ergebnis der vorherigen aufbauen. Parallelität findet stattdessen in der Bewertung statt: mehrere Reviewer gleichzeitig auf demselben Diff (19.3, 19.5, 19.9).

17. Erweiterte Java Enterprise Praxisbeispiele

Versionshinweis (v2.0): Wie in Teil III gilt: Die Beispiele sind gegen Java 21 LTS und Spring Boot 3.x formuliert und wurden für v2.0 nicht auf Spring Boot 4 gehoben; zu den Umstellungspunkten (u. a. Jackson 3) siehe Kapitel 19. In aktuellen Camunda-8-Spring-SDKs heißt die Worker-Annotation `@JobWorker` statt `@ZeebeWorker`.

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");

    // Spring-Zeebe-SDK vor 8.1; seither (und im Camunda Spring SDK):
    // @JobWorker - siehe Versionshinweis am Kapitelanfang
    @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();
    }
}
```

Praxis-Check SoftwareFabrik (abweichend): Spring Security mit RBAC ist real umgesetzt; Lizenz-Leases sind signierte JWTs, die Attestierungsschicht signiert mit Ed25519 (19.6, 19.7). Eine Workflow-Engine wie Camunda wurde dagegen nicht gebraucht: Die Zustandsmaschine des Prozesses ist das Run-Aggregat selbst (19.3).

18. MCP-Server & Hooks

Das Model Context Protocol (MCP) war bei Erscheinen der v1.3 ein Anthropic-Protokoll zur Werkzeuganbindung. Diese Einordnung ist überholt: MCP — im November 2024 von Anthropic vorgestellt [6] — wurde im Dezember 2025 an die Agentic AI Foundation unter dem Dach der Linux Foundation übergeben [8] und wird dort herstellerneutral weiterentwickelt [10]; die Foundation wurde von Anthropic, Block und OpenAI mitgegründet und wird unter anderem von Google, Microsoft und AWS getragen (Stand August 2026) [7]. Wer heute Werkzeuganbindung für Agenten baut, baut sie gegen einen Industriestandard, nicht gegen ein Vendor-Protokoll — dieselbe Entwicklung, die `AGENTS.md` für die deklarative Konfiguration genommen hat (Kapitel 4).

MCP-Server Konfiguration

Projektweite MCP-Server werden bei Claude Code in einer `.mcp.json` im Repository-Root konfiguriert (Stand August 2026 [4]; v1.3 nannte hier noch `.claude/settings.json` — dort liegen heute Hooks und Berechtigungen). Servernamen und -pakete im Beispiel sind teils fiktiv; der frühere Referenzserver `@modelcontextprotocol/server-postgres` ist seit Juli 2025 archiviert und ungepflegt (dokumentierte SQL-Injection-Schwachstelle) — für den produktiven Einsatz ist ein gepflegter Server zu wählen; das Beispiel steht hier nur für die Struktur:

```
// .mcp.json (Repository-Root)
{
    "mcpServers": {
        "postgres-server": {
            "command": "npx", "args": ["-y",
                    "@modelcontextprotocol/server-postgres"],
            "env": { "DATABASE_URL": "${DATABASE_URL}" }
        },
        "jira-server": {
            "command": "npx", "args": ["-y", "firmeninternes-jira-mcp-paket"],
            "env": {
                "JIRA_URL": "https://firma.example",
                "JIRA_TOKEN": "${JIRA_API_TOKEN}"
            }
        }
    }
}
```

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	Hauptagent beendet seine Antwort	Abschlussprüfungen, Weiterarbeit erzwingen
SessionEnd	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);
    }
}
```

Praxis-Check SoftwareFabrik (bewusst offen): MCP ist dort nicht umgesetzt. Das funktionale Äquivalent ist die versionierte, mandantengescopte Skill-/Plugin-Bibliothek plus die Materialisierung von Agenten-Definitionen und Skills je Run in den Workspace — mit dem Governance-Vorteil, dass nachweisbar bleibt, welche Erweiterung in welcher Version im Kontext des Agenten war (19.6).

Teil VI.

Fallstudie: Die SoftwareFabrik

19. Die SoftwareFabrik — von der Referenzarchitektur zum System

Hinweis: Die Konzeptkapitel dieses Whitepapers — Kapitel 1–18 sowie die Architekturentscheidungen (Kapitel 20) und der Werkzeugvergleich (Kapitel 21) — beschreiben die Referenzarchitektur, wie sie in Version 1.3 (März 2026) entworfen wurde. Dieses Kapitel beschreibt, was daraus wurde: die *Agentic Software Factory* (Produktname: SoftwareFabrik), ein vom Autor gebautes und betriebenes System, das die Referenzarchitektur produktisiert. Es ist ein Erfahrungsbericht aus erster Hand — alle Angaben sind aus dem Quellcode des Systems erhoben (Erhebungsstand 9. August 2026 auf dem Release-Stand 0.30.0), nicht aus Projektdokumentation oder Erinnerung. Wo eine Aussage im Code verankert ist, nennt eine Fußnote die konkrete Klasse; die Klassennamen dienen der präzisen Verortung — das Repository des Systems ist derzeit nicht öffentlich.

Stand und Zielbild: Dieses Kapitel unterscheidet zwischen dem implementierten Stand der SoftwareFabrik (Abschnitte 19.1–19.9 — alles dort Beschriebene ist implementiert und in Betrieb, Stand 0.30.0) und ihrer Weiterentwicklung (Abschnitt 19.10 — dort ist der Umsetzungsstand je Stufe ausgewiesen: Die Stufen 0 bis 4 sowie die Kennzahlenmessung der Stufe 6 sind hinter einem standardmäßig deaktivierten Feature-Flag umgesetzt, von Stufe 5 die Koordinationsschicht auf einem Host; allein der verteilte Betrieb über mehrere Hosts ist bewusst zurückgestellt. Welches Release welche Stufe brachte, zeigt die Release-Verlaufstabelle in 19.10; zwei Ausnahmen wirken auch ohne Feature-Flag: der Einzel-Run-Teil der Rollback-Erkennung und das Aufwandsfeld an der Run-Freigabe). Der aktuelle Stand verwendet genau einen schreibenden Agenten je Run und mehrere unabhängige Read-only-Reviewer. Diese Entscheidung entstand aus der praktischen Erfahrung, dass mehrere gleichzeitig schreibende Agenten Konfliktkosten, unklare Zurechnung und schwer attestierbare Zwischenstände erzeugen. Die nächste Ausbaustufe verwirft das parallele Agentenmodell nicht, sondern ordnet es neu: Parallelität wird auf eine übergeordnete Workflow-Ebene verlagert, das Single-Writer-Prinzip bleibt je Workspace bestehen. Diese Verlagerung hat mit 0.21.0 an der risikoärmsten Stelle begonnen. Die Beschreibung des Zielbilds ist damit keine Behauptung über den aktuellen Produktstand, sondern eine nachvollziehbare Roadmap aus der bestehenden Architektur heraus.

19.1. Von der Referenzarchitektur zur Implementierung

Die SoftwareFabrik ist eine **Control Plane für agentische Softwareentwicklung**. Sie schreibt keinen Code selbst und hostet kein Sprachmodell. Sie steuert, begrenzt, protokolliert und bewertet die Arbeit externer Coding-Agenten — und macht deren Ergebnis prüfbar.

Der Unterschied zur direkten CLI-Nutzung eines Coding-Agenten ist genau der Unterschied zwischen *einem Entwickler mit einem mächtigen Werkzeug* und *einem Entwicklungsprozess*: Spezifikation, Freigabe, Isolation, Review, Nachvollziehbarkeit, Budget, Mandantentrennung. In einem Satz: Die SoftwareFabrik ist die produktisierte Umsetzung der in diesem Whitepaper beschriebenen Referenzarchitektur — vendor-neutral statt an ein einzelnes Werkzeug gebunden, und um die Governance-Schicht erweitert, die in regulierten Umfeldern verlangt wird (19.6).

Der Kernablauf

Ein Vorhaben durchläuft die Fabrik in sechs Schritten:

1. **Erfassen.** Ein sechsschrittiger Wizard sammelt Projektidee, Zielplattform, Backend- und Frontend-Stack; 18 Templates decken Web, Mobile und Desktop ab, inklusive Import eines bestehenden Repositories.
2. **Spezifizieren.** Daraus generiert die Plattform versionierte Markdown-Artefakte (`PROJECT.md`, `INSTRUCTIONS.md`, `AGENTS.md`, `WORKFLOW.md`, `DEFINITION_OF_DONE.md`, `README.md`), die der Mensch vor dem Lauf editieren kann. Das ist der Punkt, an dem menschliche Absicht maschinenlesbar wird.
3. **Ausführen.** Ein *Run* durchläuft sieben Phasen. Der Coding-Agent läuft in einer Sandbox auf einem projektpersistenten Workspace mit eigenem Git-Repository, auf einem eigenen Branch.
4. **Prüfen.** Read-only-Review-Adapter analysieren den Diff; ein Quality Gate aggregiert die Befunde zu PASS/WARN/FAIL (bzw. ERROR bei Prüferausfall). Bei Fehlschlag speist die Plattform das Feedback zurück in den Agenten — bis zu zwei automatische Korrekturversuche.
5. **Übergeben.** Erfolgreiche Runs werden lokal gemergt oder als Pull Request gepusht; bei aktivierter PR-Rückkopplung wartet der Run auf grüne CI und den Merge.
6. **Belegen.** Jeder relevante Schritt landet in einer signierten, append-only Audit-Hashkette; ein Warum-Trace rekonstruiert für jeden Run, *welche* Policy, *welches* Modell und *welche* Freigabe gewirkt haben.

Kennzahlen der Codebasis

Kennzahl	Wert
Produktivklassen (Java)	421
Produktivcode	~41.700 Zeilen
Testklassen	291
Fachliche Slices (Module)	30

Kennzahl	Wert
Datenbanktabellen	58
Flyway-Migrationen	52
Execution-Adapter	10
Review-Adapter	6
Wizard-Templates	18
Coverage-Gate	Line $\geq$ 85 %, Branch $\geq$ 81 % (JaCoCo, buildbrechend)
Releases	38 (0.1.0 bis 0.30.0, April–August 2026)

Erhoben auf dem Release-Stand 0.30.0 (9. August 2026). Der Zuwachs der Releases 0.21.0 bis 0.30.0 geht fast vollständig auf die Workflow-Ebene zurück — welches Release welche Roadmap-Stufe brachte (einschließlich der Migrationen V40–V54), dokumentiert die Release-Verlaufstabelle in 19.10. Die 30. Fachlichkeit ist die Kennzahlenmessung (19.10).

Der Technologiestack ist bewusst konservativ: Java 25, Spring Boot 4.0, server-gerendertes UI (Thymeleaf + HTMX, Server-Sent Events für Live-Logs, kein SPA), PostgreSQL 18 mit Flyway, Docker Compose. Ein Maven-Modul, ein Prozess, eine Datenbank — kein Cluster, kein Message-Broker, kein Service-Mesh. Die Komplexität des Systems liegt in der *Steuerung von Nichtdeterminismus*, nicht in der Infrastruktur, und genau dort soll sie auch bleiben. Zwei Umstellungspunkte des Sprungs auf Spring Boot 4 sind für Nachbauer erwähnenswert: Jackson 3 (`tools.jackson`) ist dort der Standard — auto-konfiguriert wird ein Jackson-3-`JsonMapper`; Jackson 2 (`com.fasterxml`) bleibt zwar im Dependency-Management, wird aber nicht mehr auto-konfiguriert, neuer Serialisierungscode muss also den Jackson-3-Bean injizieren oder einen eigenen Mapper mitbringen —, und Signaturen über persistierte Daten brauchen auf Millisekunden trunkierte Zeitstempel (19.6).

Bewusste Nicht-Ziele

Für die Einordnung ist die Abgrenzung so wichtig wie der Funktionsumfang:

- **Kein eigenes Modell-Hosting.** Die Fabrik ist Steuerschicht, kein Inferenz-Stack. Lokale Modelle werden über eine CLI (z. B. Ollama) angebunden.
- **Keine Web-IDE.** Entwickelt wird weiter in IntelliJ oder VS Code; die Fabrik besitzt den Prozess, nicht den Editor.
- **Kein Consumer-SaaS.** Zielbild ist die Installation im Unternehmen — bis hin zum Air-Gap-Betrieb.
- **Kein Real-Time-Co-Editing**, kein eigener Build-Server, kein Kubernetes-Zwang.

19.2. Systemkontext und Bausteinsicht

Der Systemkontext unterscheidet drei menschliche Rollen — Architekt/Lead Developer (spezifiziert, gibt frei), Auditor/Compliance (liest Nachweise) und Administrator (Mandanten, Rollen, Policies) — und drei Klassen externer Systeme: die Coding-Agenten als Subprozesse (Vendor-CLIs und Cloud-Gateways), das Zielrepository mit seiner CI, und die Scanner- und Lizenzinfrastruktur.

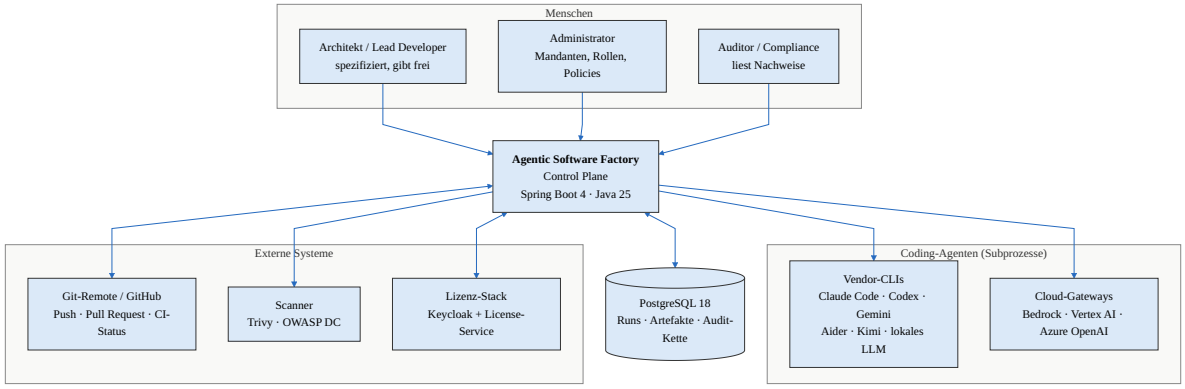

Abbildung 19.1.: Systemkontext der Agentic Software Factory: eine Control Plane zwischen Mensch, Coding-Agenten und Zielrepository

Gegenüber dem Systemkontext aus Kapitel 3 fällt auf: Auditor und Administrator sind eigenständige Akteure geworden. Das ist kein Zufall, sondern die Konsequenz aus dem Governance-Anspruch — wer Nachweise verlangt, braucht eine Rolle, die sie liest.

Modularer Monolith mit erzwungenen Grenzen

Die Fabrik ist ein **modularer Monolith**: ein Maven-Modul, ein Deployment-Artefakt, je ein sauber geschnittenes Paket (*Slice*) pro Bounded Context — insgesamt 30. Jeder Slice trägt seine eigenen Schichten (`domain`, `application`, `web`, teils `infrastructure`); externe Systeme — Coding-CLIs, Git, Maven, Scanner, GitHub-API — sitzen ausschließlich hinter Ports.

Die Größenverteilung der Slices ist selbst eine Aussage: `run` und `execution` machen zusammen ein Viertel der Codebasis aus — die Steuerung des nichtdeterministischen Teils ist der eigentliche Aufwand, nicht die Fachlichkeit drumherum. Die Governance-Slices (`policy`, `audit`, `mandant`, `provenance`, `export`, `skills`) sind dagegen klein: Governance kostet vor allem *Entwurfsdisziplin*, nicht Code.

Der entscheidende Unterschied zum Konzeptteil dieses Whitepapers: Die Architekturregeln sind nicht beschrieben, sondern **maschinell erzwungen**. Drei ArchUnit-Testklassen laufen in jedem Build:[1]

- **Schichtenregeln:** `domain` hängt weder von `web` noch von der Execution-Infrastruktur ab; `application` nicht von `web`; Controller nur in `..web..`; kein `JpaRepository` in `..web..` oder `..application...`
- **Hexagonale Regeln:** `domain` kennt `application` nicht; weder `application` noch `web` kennt eine konkrete `ExecutionAdapter`-Implementierung. Die letzten beiden Regeln sind der Kern der Vendor-Neutralität: Auf Klassenebene ist es ausgeschlossen, versehentlich einen Service an einen konkreten Vendor zu koppeln — der Build bricht.
- **Debt-Ratchet:** Zwei Regeln, die heute *nicht* eingehalten werden (Modulzyklen; 13 direkte `web` → `JpaRepository`-Zugriffe), werden mit ArchUnits `FreezingArchRule` als versionierte Baseline eingefroren. Ein *neuer* Verstoß bricht den Build; ein *behobener* verschwindet automatisch aus der Baseline. Die Schuld ist gezählt und sichtbar statt implizit.

[1] `LayeringRulesTest`, `HexagonalRulesTest`, `ArchitectureDebtRatchetTest` im Testbaum des Systems.

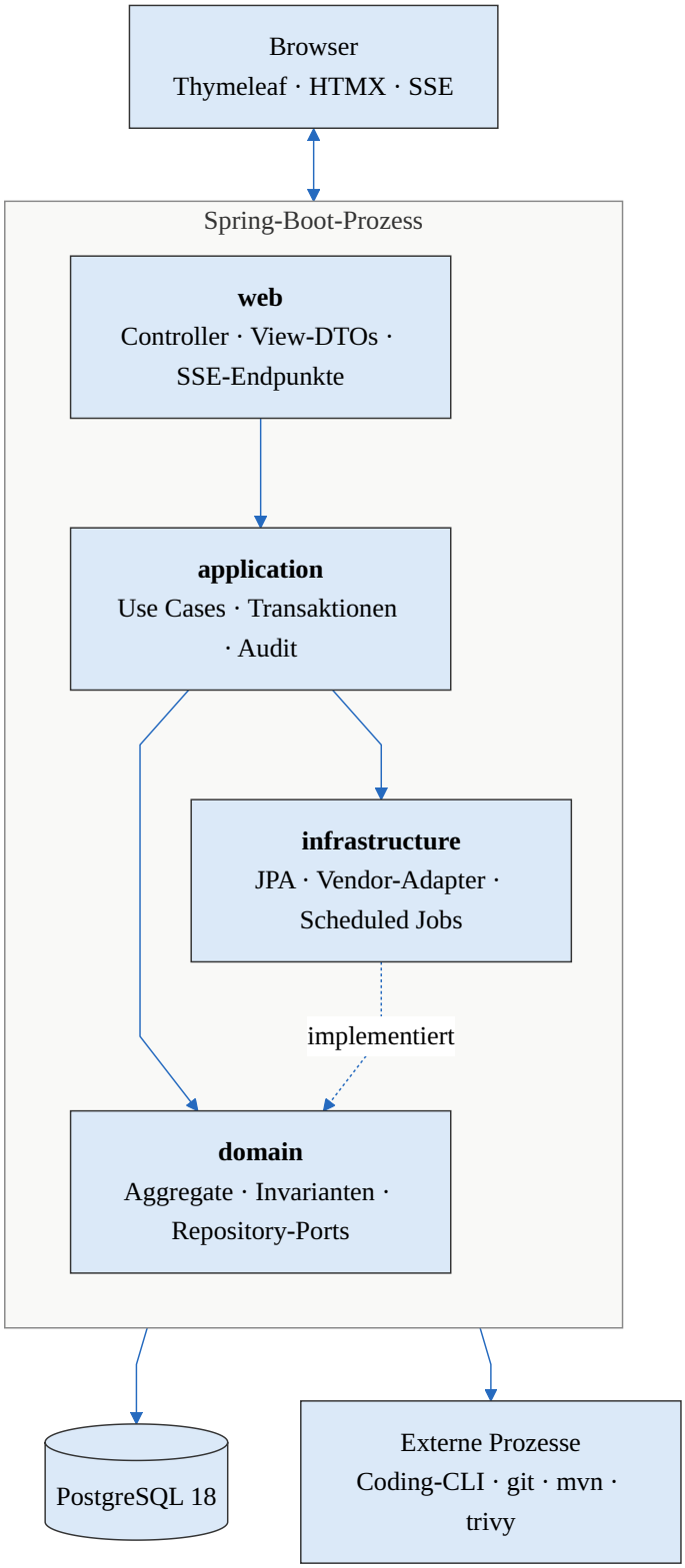

Abbildung 19.2.: Bausteinsicht: modularer Monolith mit Ports-and-Adapters pro Slice; externe Werkzeuge ausschließlich hinter Ports

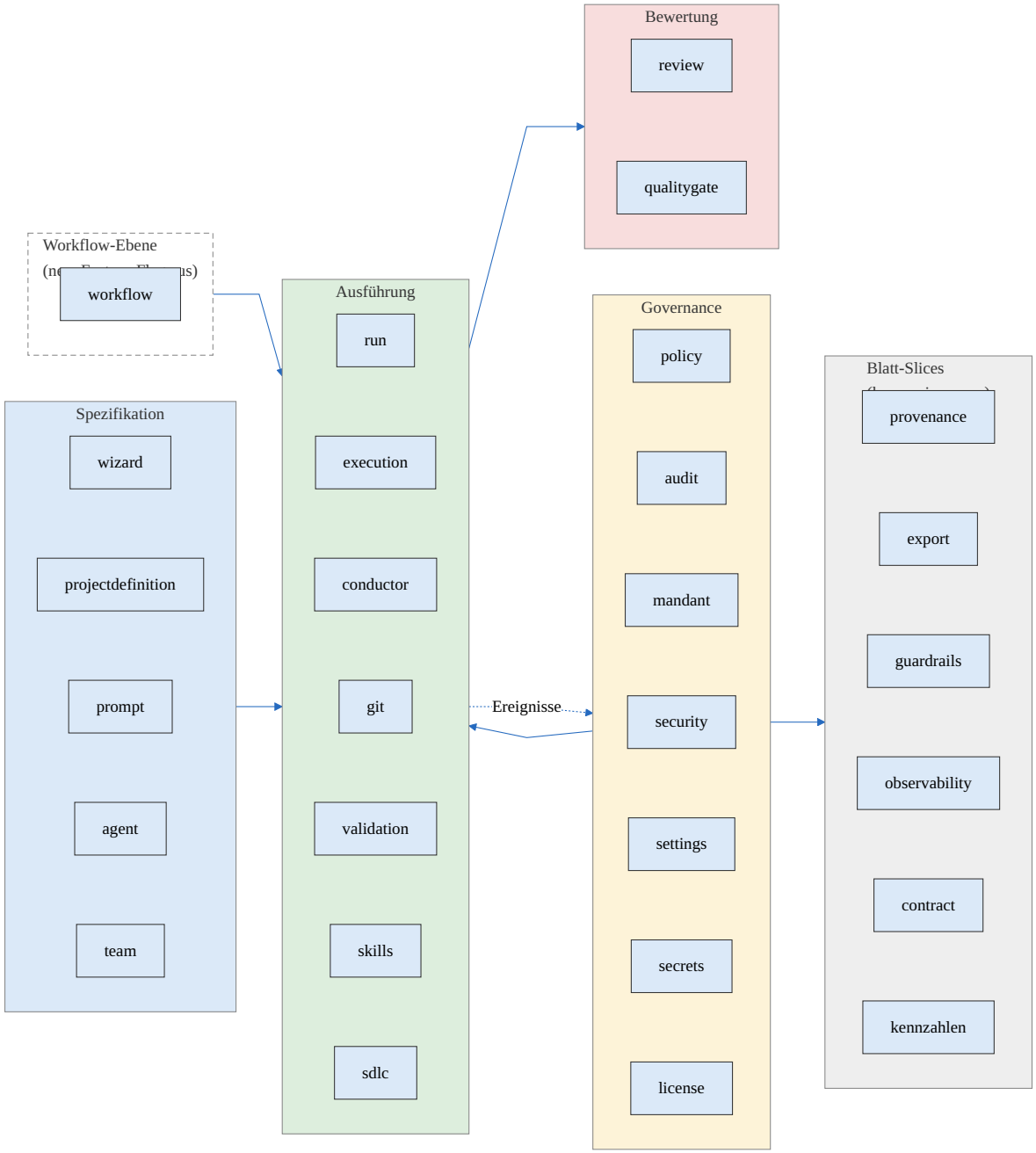

Abbildung 19.3.: Fachliche Slices der Fabrik, gruppiert nach Aufgabe. Blatt-Slices (provenance, export, guardrails, observability, contract, kennzahlen) konsumieren nur; die technischen Querschnittspakete common und web sind nicht dargestellt

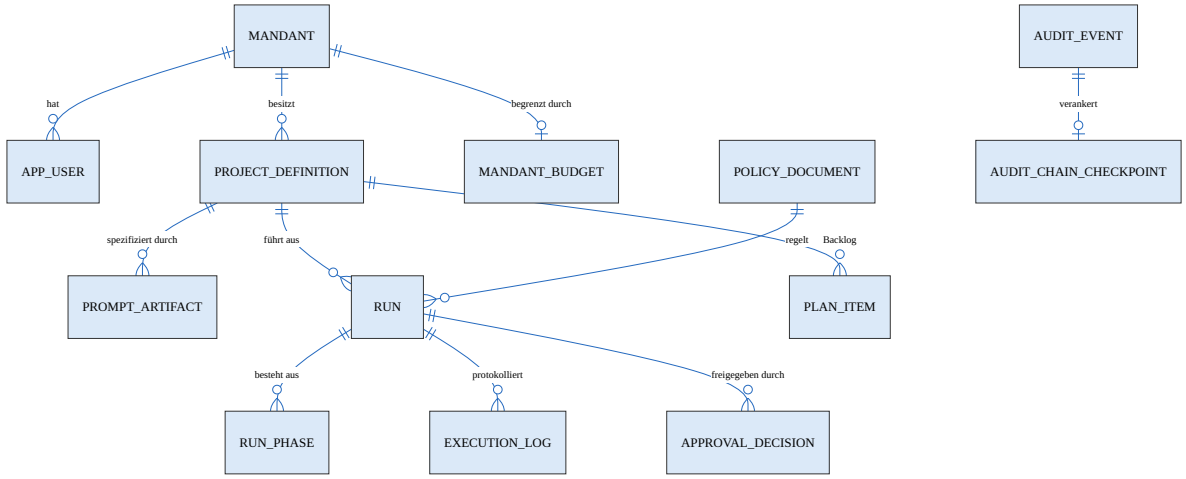

Abbildung 19.4.: Kernaggregate der Run-Ebene; vollständig umfasst das Schema 58 Tabellen in 52 Flyway-Migrationen

Der Ratchet hat einen lehrreichen Fallstrick: Der Freeze speichert nur eine begrenzte Zahl elementarer Modulzyklen (Kappungsgrenze, Standard 100). Solange die tatsächliche Zyklenzahl darunter liegt, ist die Enumeration vollständig und deterministisch. Überschreitet eine neue Cross-Slice-Kante die Grenze, wird die Enumeration abgeschnitten und *reihenfolgeabhängig* — der Test wird lokal grün und in der CI rot, ohne dass sich der Code inhaltlich unterscheidet. Das ist ein reales Betriebsrisiko automatisierter Architekturmetriken und ein Beleg für eine These, die sich durch dieses Kapitel zieht: **Guardrails müssen selbst gewartet werden.**

Ein wiederkehrendes Entwurfsmuster verdient Erwähnung, weil es aus dem Ratchet folgt: Neue Querschnittsfähigkeiten werden konsequent als **Blatt-Slice** angelegt — ein Paket, das nur konsumiert und von niemandem konsumiert wird. Der Warum-Trace (`provenance`) korreliert Run-, Metrik-, Plan-, Freigabe- und Audit-Daten, ohne dass irgendein anderer Slice davon abhängt; der Audit-Export (`export`) aggregiert dasselbe zum Bundle. Eine Fähigkeit, die naturgemäß überall hinschaut, würde als zentraler Service sofort Modulzyklen erzeugen — als Blatt-Slice erzeugt sie keinen einzigen.

Das Domänenmodell: der Entwicklungsprozess als Fachdomäne

Die Fabrik modelliert **den Entwicklungsprozess selbst** als Domäne. Nicht der generierte Code ist das Fachobjekt, sondern Spezifikation (`ProjectDefinition`, `PromptArtifact`), Lauf (`Run` mit `RunPhase`), Backlog (`PlanItem` mit Abhängigkeiten), Freigabe (`ApprovalDecision`), Regelwerk (`PolicyDocument`) und Nachweis (`AuditEvent`). Das ist die konsequente Anwendung von Kapitel 11 auf das Agentensystem selbst — mit einer bewussten Pragmatik: Domänenaggregat und JPA-Entität sind dieselbe Klasse. Kein separates Persistenzmodell, kein Mapping-Code; die Entscheidung ist im Code dokumentiert und in den ArchUnit-Regeln bewusst ausgespart, statt sie stillschweigend zu brechen. Für die reine DDD-Lehre ist das ein Verstoß; für ein System, dessen Komplexität woanders liegt, ist es eine bewusste Abwägung zugunsten der Umsetzungsgeschwindigkeit.

Der Migrationsverlauf liest sich als Reifungskurve des Systems: V1–V9 Grundschema (Werk-

zeug), V12–V18 Wizard und Projektgedächtnis (Prozess), V19–V25 Plan-/Build-Runs, Branches, Quality Gate (Lebenszyklus), V26–V33 ausschließlich Mandanten- und Nachweisstrukturen (Mehrmandantenfähigkeit und Nachweisfähigkeit), V34–V39 Repository-Realität, Skills, Routinen, V40–V45 die Workflow-Ebene und damit die Parallelität, V46–V51 Vertragsregistrierung, versionierte Planänderungen, Konfliktklassifikation und Worker-Ansprüche, V52 die Abdeckungswerte am Build-Ergebnis, V53 die Rollback-Anker an Merge-Queue und Run, V54 das freiwillige Aufwandsfeld an der Freigabeentscheidung. Auf diese Kurve kommt Abschnitt 19.8 zurück.

19.3. Das Ausführungsmodell: der Run

Der *Run* ist die Ausführungseinheit der Fabrik — das Gegenstück zum Execution Model aus Kapitel 7, jedoch nicht als Task-Graph, sondern als **zustandsbehaftete Pipeline mit Regelkreis**. Die gesamte Orchestrierung hat genau eine Stelle, an der Run-Statuswechsel stattfinden; die erlaubten Übergänge liegen zentral, unerlaubte Übergänge werfen eine Ausnahme statt still zu passieren.[2]

Zwei Run-Arten, sieben Phasen, 13 Zustände

Es gibt zwei Run-Arten: Ein **Plan-Run** analysiert den Ist-Stand und schlägt die nächsten Arbeitsschritte vor; nur Dateien unter `plans/` werden committet, alles andere wird verworfen, und die Vorschläge landen als Backlog-Elemente (`plan_item`, mit Abhängigkeiten) in der Datenbank. Ein **Build-Run** setzt ein Backlog-Element tatsächlich um: Code auf einem isolierten Branch, validiert, gemergt oder als Pull Request übergeben. Die Asymmetrie ist Absicht — ein Plan-Run darf keinen Code ändern und durchläuft kein Build-Gate. Damit ist Planung ohne Schreibrisiko am Code automatisierbar, die Grundlage für zeitgesteuerte Routinen und automatische Folgevorschläge.

Jeder Run durchläuft sieben Phasen (Intake, Prompt-Assembly, Workspace-Preparation, Execution, Validation, Correction, Completion) und bewegt sich durch 13 Zustände. Drei davon tragen die Argumentation dieses Whitepapers weiter:

- `WAITING_FOR_APPROVAL` — der Mensch ist ein *Zustand im System*, kein Nebenprozess. Human Authority (Prinzip AP-6) ist damit nicht Appell, sondern Zustandsmaschine.
- `NEEDS_CORRECTION` — Fehlschlag ist ein regulärer Zustand mit definiertem Ausgang, nicht ein Abbruch.
- `WAITING_FOR_PR` — die Realität des Zielrepositories (fremde CI, fremde Reviewer, fremder Merge-Zeitpunkt) ist im Modell abgebildet, statt am Systemrand zu enden.

Ablauf eines Build-Runs

Vier Stationen des Ablaufs sind konzeptionell bemerkenswert:

Policy-Prüfung zweimal. Vor der Anlage prüft der Orchestrator Modell-Policy, aktive Policy-as-Code (ist der Adapter erlaubt?), Attestierungspflicht und Lizenzgrenzen — abgelehnte Läufe

[2] `RunOrchestrationService` (~1.580 Zeilen), `RunStatusTransitions`; unerlaubte Übergänge werfen `InvalidRunStateTransitionException`.

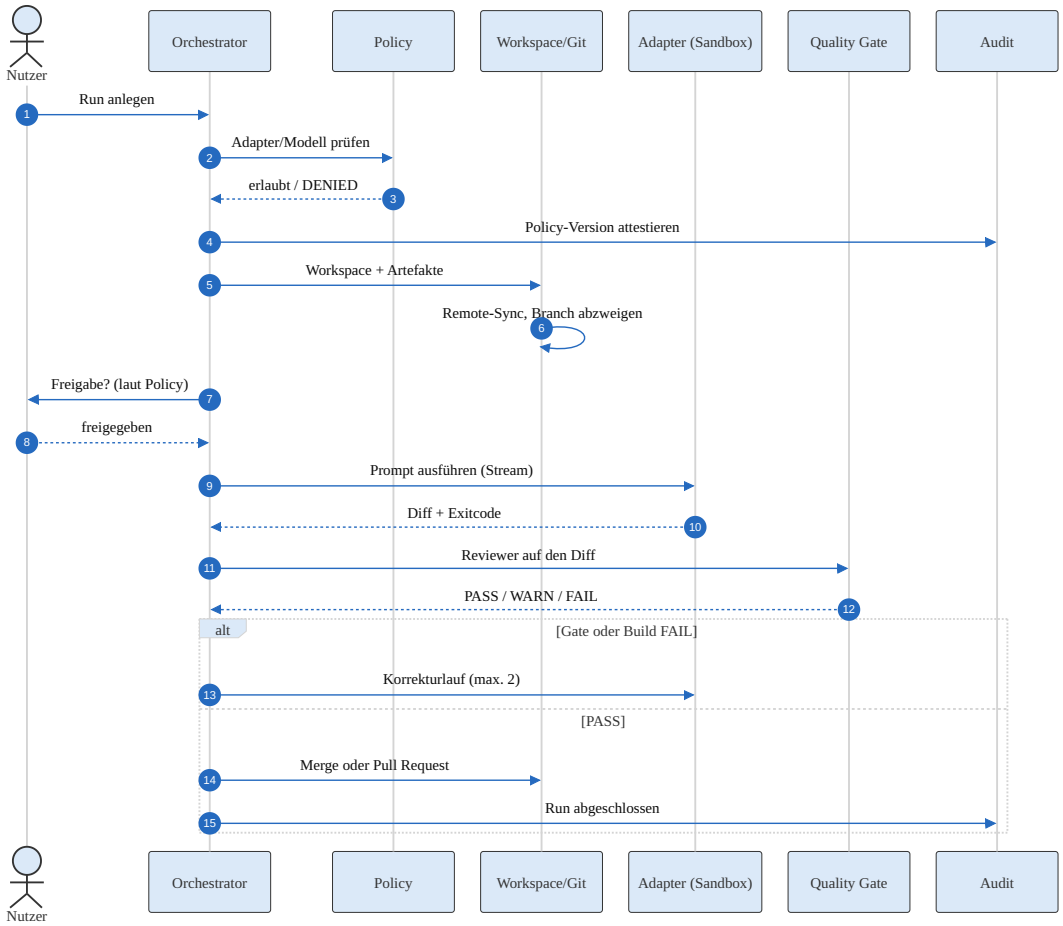

Abbildung 19.5.: Laufzeitablauf eines Build-Runs von der Anlage bis zum Merge, inklusive Korrekturschleife und Approval-Punkten

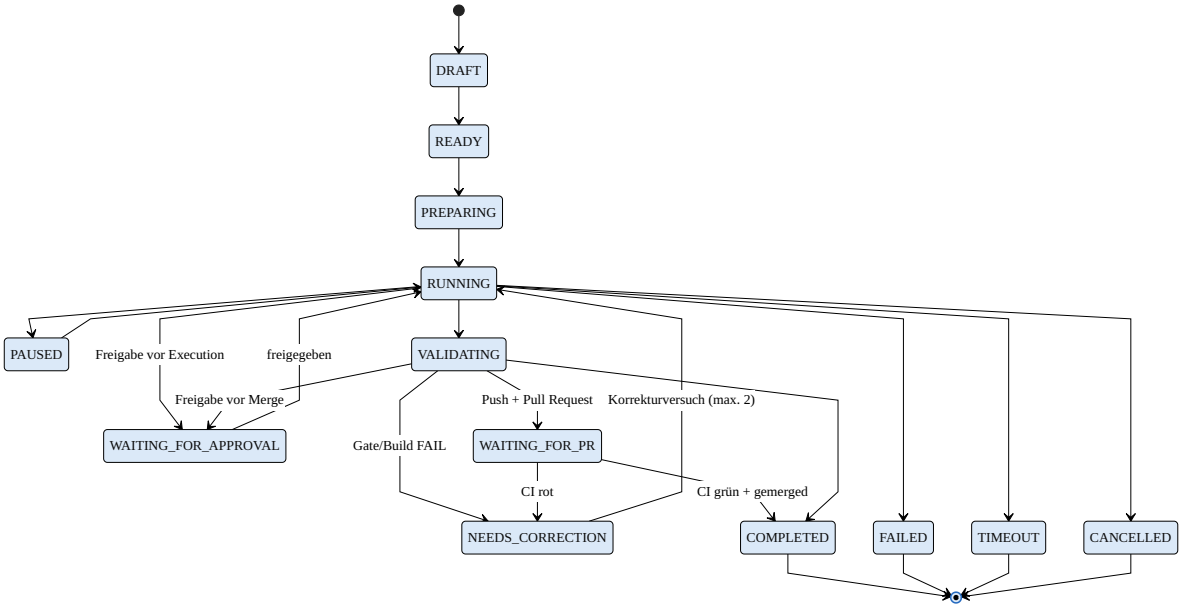

Abbildung 19.6.: Zustandsmodell eines Runs. Alle Übergänge sind zentral hinterlegt; unerlaubte Übergänge werfen eine Ausnahme

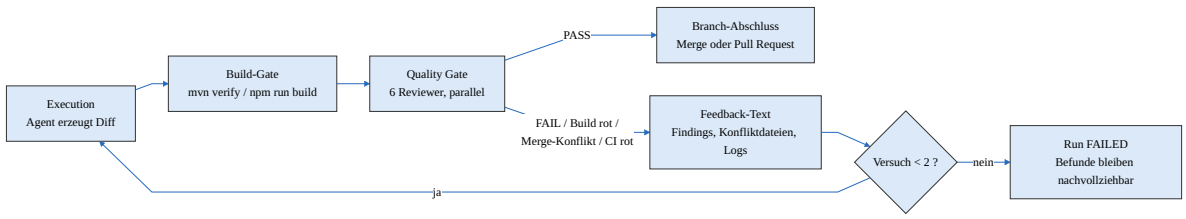

Abbildung 19.7.: Die Korrekturschleife als Regelkreis — Befunde werden zu Eingaben des nächsten Laufs

erzeugen ein attestiertes Ereignis `RUN_POLICY_DENIED`, auch die Ablehnung ist Nachweis. Beim tatsächlichen Start werden Attestierungspflicht und geltende Policy-Version *erneut* ausgewertet und gestempelt. Der Grund: Ein vor der Policy-Aktivierung angelegter Run würde sie sonst umgehen — und Warum-Trace wie Audit-Export sollen die real durchgesetzte Version ausweisen, nicht die zum Anlagezeitpunkt gültige.

Projektpersistenter Workspace, Branch-Isolation. Der Workspace gehört dem Projekt, nicht dem Run; der erste Lauf initialisiert Git, Folgeläufe arbeiten auf dem bestehenden Code weiter. Bei Remote-Projekten wird der Base-Branch *vor* dem Abzweig auf den Remote-Stand vorgespult, damit der Lauf nicht auf veraltetem Code aufsetzt; dann zweigt der Run auf einen eigenen Branch (`sdlc/run-<id>`) ab. In den Workspace werden die versionierten Spezifikations-Artefakte, das kuratierte Projektgedächtnis (`MEMORY.md`) und die Guardrails-Projektion geschrieben (dazu 19.6).

Zwei Approval-Punkte. Verlangt die Policy eine Freigabe, stoppt der Run vor der Execution — und regulierte Profile verlangen ein zweites Vier-Augen-Gate vor dem Merge. Die Fortsetzung nach der zweiten Freigabe führt *nicht* erneut aus, sondern finalisiert nur den bereits validierten Stand; sonst wäre die signierte Zusage Fassade.

Abschluss an der Repository-Realität. Entweder lokaler Merge oder Push mit Pull Request; im PR-Fall wartet der Run auf grüne CI und den Merge. Das zugehörige Backlog-Element wird erst mit dem Merge auf `DONE` gesetzt — nicht mit dem Codeschreiben; mehrere abgeschlossene Runs lassen sich als **Meilenstein-Release** mit Changelog, Tag und Release bündeln. Ein abgeschlossener Build-Run kann über ein Ereignis einen Folge-Plan-Run anstoßen; zusammen mit zeitgesteuerten Routinen entsteht der geschlossene Kreis *planen* → *bauen* → *prüfen* → *mergen* → *neu planen*, mit definierten Stellen, an denen ein Mensch eingreifen muss.

Der Regelkreis: Korrekturschleife statt Retry

Kapitel 7 beschreibt eine Retry-Strategie. Die Praxis hat daraus einen **Regelkreis** gemacht: Vier Ereignisse speisen dieselbe Schleife — Build fehlgeschlagen, Quality Gate blockiert, **Merge-Konflikt** mit dem Base-Branch, rote CI am Pull Request. In allen vier Fällen wird ein Feedback-Text erzeugt (Build-Ausgabe, Findings, Konfliktdateien) und als zusätzlicher Kontext in einen erneuten Agentenlauf eingespeist — maximal zwei Versuche, danach bleibt der Run in `NEEDS_CORRECTION` und die Befunde bleiben nachvollziehbar.

Zwei Details aus dem Betrieb:

- **Merge-Konflikte gelten als Arbeitsanweisung, nicht als Systemfehler.** Der Konflikt

wird dem Agenten mit den betroffenen Dateien und der expliziten Aufforderung übergeben, die Konfliktmarker aufzulösen. Repository-Realität wird damit Teil der Aufgabe.

- **Der Branch-Wechsel vor der Korrektur ist sicherheitsrelevant.** Nach einem abgebrochenen Merge steht der Workspace auf dem Base-Branch. Ohne expliziten Rückwechsel würde die Korrektur samt `git add -A` direkt auf `main` committen — und damit sowohl die Branch-Isolation als auch die Pflichtfreigabe umgehen. Dieser Fall wurde in einem adversarialen Sicherheits-Re-Review gefunden und behoben. Er illustriert eine Kernerfahrung: Agentische Systeme scheitern an den *Übergängen* zwischen Automatismen, nicht in ihnen.

19.4. Vendor-Neutralität als Architektureigenschaft

Kapitel 21 (in Version 1.3: Kapitel 20) vergleicht Werkzeuge gegeneinander. Die Fabrik hat die Frage architektonisch aufgelöst: Die gesamte Vendor-Neutralität hängt an einer Schnittstelle,[3] hinter der zehn Adapter stehen — ein deterministischer `mock`-Adapter (Default; macht das System ohne Vendor demonstrierbar und testbar), sechs CLI-Adapter (Claude Code, Codex, Gemini, Aider, Kimi, lokales LLM z. B. via Ollama) und drei Cloud-Gateways (AWS Bedrock, Google Vertex AI, Azure OpenAI). Die Adapterwahl ist eine Konfigurationsfrage je Run, aufgelöst über eine Hierarchie (Run-Override > Projekt-Default > User-Setting > globales Setting > Konfigurationsdatei); ein Projekt kann die erlaubten Adapter einschränken, und Policy-as-Code kann diese Wahl mandantenweit übersteuern.

Drei Entwurfsentscheidungen im Port selbst:

1. **Streaming statt Rückgabewert.** Ein Event-Consumer liefert Logzeilen, Token-Verbrauch und Phasensignale *während* des Laufs — ohne das wäre Live-Beobachtbarkeit unmöglich und ein Abbruch bliebe folgenlos.
2. **Verfügbarkeit ist Teil des Vertrags.** Adapter, deren CLI fehlt, verschwinden geordnet aus der Auswahl, statt zur Laufzeit zu scheitern.
3. **Timeout ist ein eigener Ergebniszustand**, nicht ein Sonderfall von Fehler. Bei nichtdeterministischen Agenten ist „hat zu lange gebraucht“ fachlich etwas anderes als „ist gescheitert“.

Einschränkung: Die drei Cloud-Gateway-Adapter sind konfigurierbar und degradieren sauber, wurden aber nicht end-to-end gegen echte Cloud-Credentials verifiziert — sie sind als „vorbereitet“, nicht als „erprobt“ zu bezeichnen.

Abo-Modus statt Token-Abrechnung

Ein praktisch sehr relevanter Punkt: Coding-CLIs lassen sich meist auf zwei Wegen authentifizieren — per API-Key (Abrechnung je Token) oder per Abo-Login (Flatrate). Die Fabrik unterstützt für Claude Code, Codex und Kimi beides explizit. Der kritische Teil ist das **aktive Entfernen des API-Keys aus der Subprozess-Umgebung im Abo-Modus**: Sonst würde die CLI stillschweigend den kostenpflichtigen Pfad wählen — ein Fehler, der erst auf der Rechnung sichtbar wird. Die Konsequenz für das Kostenmodell aus Kapitel 15 ist grundlegend: Bei Flatrate-Abos

[3] `ExecutionAdapter` im `execution`-Slice.

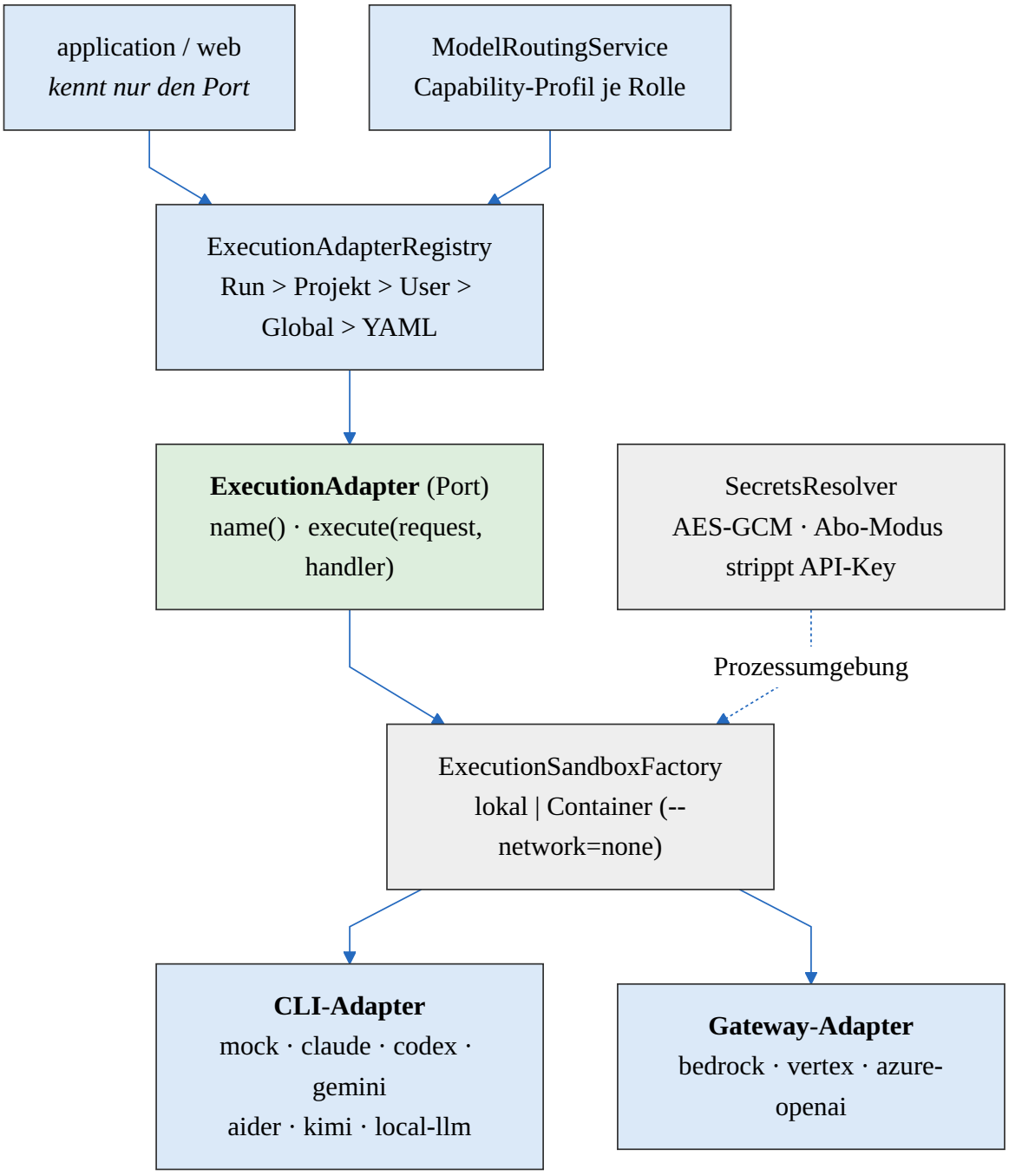

Abbildung 19.8.: Vendor-Neutralität durch einen Port: Application- und Web-Schicht kennen ausschließlich den ExecutionAdapter (ArchUnit-erzwungen)

entstehen keine Token-Kosten je Lauf; ein reines Token-ROI-Modell bildet die Wirtschaftlichkeit agentischer Entwicklung nicht mehr vollständig ab.

Flankiert wird das von einem vollständigen Kostenmodell in der Plattform: eine Preistabelle je Modell mit getrennten Preisen für Input, Output und Cached Input, Kostenaggregation nach Projekt, Run, Provider, Mandant und auslösendem Nutzer (*Seat*), harte Budget-Caps je Mandant sowie Tages- und Wochenlimits mit Soft-Schwelle. Für Modelle ohne hinterlegte Preisdaten weist das System keine Kosten aus. Das ist eine bewusste Entscheidung gegen geratene Preise — aber, wie ein Review zu Recht angemerkt hat, keine konservative Budgetbewertung: Ein mit 0 € bewerteter Lauf kann Budgetgrenzen unterlaufen und Kosten zu niedrig ausweisen. Sauberer wäre ein expliziter Kostenstatus `UNKNOWN`, der die Budgetprüfung nicht bestehen kann, ein konfigurierbarer Sicherheitsersatzwert — und im Abo-Modus ein eigener Status `FLAT_RATE` statt der Zahl null. Ein Punkt für die Weiterentwicklung (19.10).

Capability-Routing statt Modellnamen

Die Modellwahl-Strategie aus Kapitel 5 ist umgesetzt — aber über **Fähigkeitsprofile** statt hart verdrahteter Modellnamen: Eine Routing-Rolle (Planung vs. Umsetzung) verlangt eine Fähigkeitsstufe, die zur Laufzeit auf ein konkretes Modell aufgelöst wird. Planung darf ein stärkeres, teureres Modell bekommen als mechanische Umsetzung, ohne dass irgendwo ein Modellname im Code steht — wichtig in einem Markt, in dem Modellnamen eine Halbwertszeit von Monaten haben. Eine je Projekt gesetzte Modell-Policy wird beim Anlegen eines Runs erzwungen und attestiert; im Nachhinein ist belegbar, welches Modell wirklich gearbeitet hat.

Sandbox

Eine Sandbox-Factory wählt zwischen zwei Varianten. Der Default ist die **Prozessisolation**: ein eigener Prozess je Run mit sauber gesetzter Umgebung und einer Allowlist der Variablen, die den Agenten überhaupt erreichen. Alternativ, per Einstellung aktivierbar, die **Container-Sandbox**: ein ephemerer Container je Agentenlauf mit CPU-, Speicher- und Prozesslimits, read-only-Dateisystem, Bindmount ausschließlich auf den Workspace und `--network=none` als Voreinstellung dieser Variante. Die Netzwerksperre ist eine starke Aussage: Ein Coding-Agent braucht im Normalfall keinen ausgehenden Netzzugriff — Abhängigkeiten kommen aus dem vorbereiteten Workspace beziehungsweise dem Cache. Wer das Netz öffnet, tut es bewusst.

Zur Absicherung der Außenkontakte gehört eine **Host-Allowlist für Git-Remotes**:[4] Nur für erlaubte Hosts wird der Plattform-Token verwendet und eine API-Anfrage ausgeführt. Ohne diese Grenze könnte eine manipulierte Remote-URL den fabrikweiten Token exfiltrieren; zusätzlich wird das `git ext::`-Protokoll unterbunden, das Kommandoausführung über Remote-Helper erlauben würde. Beide Befunde stammen aus einem adversarialen Re-Review — Kapitel 14 hatte die Bedrohungsklassen richtig benannt, die konkreten Angriffsflächen zeigten sich erst im gebauten System.

[4] `RemoteUrlPolicy` im `common`-Slice.

19.5. Guardrails in der Praxis

Die Guardrails-Pipeline aus Kapitel 12 und ADR-3 ist umgesetzt — mit einer architektonischen Präzisierung, die sich als tragend erwiesen hat: Es gibt einen **zweiten Schichtbegriff**. Coding-Agenten *schreiben* Code; Review-Adapter sind *read-only* und dürfen ihn nicht verändern. Ein System, in dem dieselbe Instanz erzeugt und freigibt, hat kein Gate, sondern eine Selbsteinschätzung.

Sechs Review-Adapter laufen parallel auf dem Diff: zwei LLM-Reviewer (Diff-Review durch Claude Code bzw. Aider im Read-only-Modus), drei statische Prüfer (Security-Muster wie API-Key-Präfixe, Path-Traversal, SQL-Konkatenation; Architektur-Verstöße inklusive Änderungen an bestehenden Datenbankmigrationen; Halluzinationsindikatoren) und ein werkzeuggestützter Dependency-Scan (CVEs und Lizenzverstöße; CVE-Befunde der Kategorie Security blockieren). Die Mischung ist Absicht: LLM-Reviewer finden Kontextfehler, die keine Regel beschreibt; statische Reviewer finden deterministisch, kostenlos und auditierbar genau das, was ein nichtdeterministisches Modell nicht zuverlässig findet. Ein Gate, das nur aus LLM-Urteilen besteht, wäre selbst nichtdeterministisch.

Der **Halluzinations-Reviewer** verdient besondere Erwähnung: Er prüft nicht den Code, sondern die *Behauptungen über den Code* — „alle Tests laufen durch“ bei unverändertem Testverzeichnis, unberührte Akzeptanzkriterien, Importe nicht existierender Klassen. Das ist die direkte Umsetzung der Halluzinationserkennung aus Kapitel 12 — treffender als **Claim Verification** bezeichnet —, verschoben vom Code auf die Selbstauskunft des Agenten — dort sitzt der klassische Selbstbetrug agentischer Systeme.

Die Gate-Policy

Das Quality Gate aggregiert die Befunde über eine konfigurierbare Policy[5] mit acht Stellschrauben (Blockierverhalten je Schweregrad, Confidence-Schwelle, Pflichtkategorien, Pflicht-Reviewer). Drei Sonderregeln kann **keine** Policy aufweichen:

- Security-Befunde der Stufen HIGH und CRITICAL blockieren immer.
- Architektur-Befunde der Stufe CRITICAL blockieren immer.
- Ein abgestürzter Reviewer wird zu `ERROR` materialisiert und führt zur Gesamtentscheidung `ERROR` — niemals zu einem stillen Pass.

Der letzte Punkt trägt eine Grundregel des ganzen Systems: **Ein Gate, dessen Ausfall wie Erfolg aussieht, ist schlimmer als kein Gate**, weil es Vertrauen erzeugt, das es nicht deckt. Dieselbe Logik findet sich an weiteren Stellen wieder (das Lizenz-Lease ist *fail closed*, Attestierung ohne Schlüssel ist ein Startfehler — und als Projektregel: ein CI-Job, der sich ohne Secret selbst überspringt, gilt nicht als bestanden).

Das Confidence Scoring aus Kapitel 12 ist umgesetzt — allerdings ohne den dort skizzierten AOP-Aspekt: Die Vertrauenswerte der Reviewer werden im Gate-Service zu einem Gesamtwert verrechnet und gegen die Schwelle der Policy geprüft. Ein Aspekt hätte die Nachvollziehbarkeit verschlechtert, ohne funktionalen Gewinn.

[5] `QualityGatePolicy` im `qualitygate`-Slice; vorkonfiguriert als `strict()` und `lenient()`.

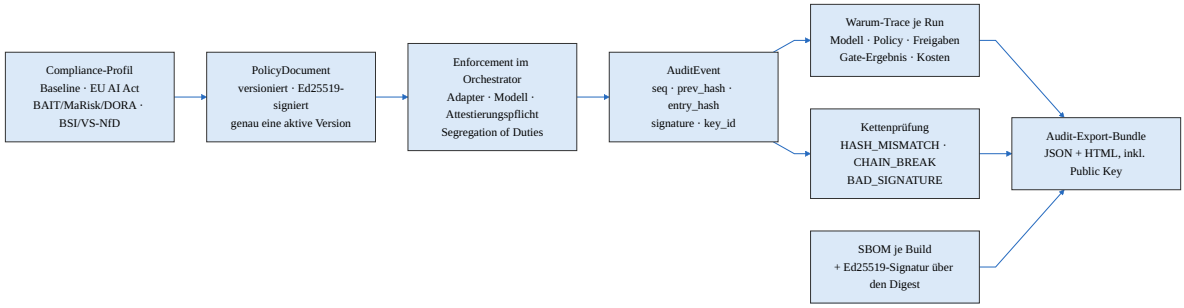

Abbildung 19.9.: Von der Policy bis zum Auditbericht: jede Durchsetzung erzeugt ein signiertes Kettenglied

Drei Betriebsmodi

Das Gate läuft in einem von drei Modi, mandantenweit steuerbar: `off`, `advisory` (Befunde werden erhoben und angezeigt, blockieren aber nicht), `blocking` (ein FAIL startet die Korrekturschleife). ADR-3 forderte das Gate als Pflicht; die Praxis hat den `advisory`-Modus ergänzt, denn der Einführungspfad in Organisationen verläuft erfahrungsgemäß so: erst messen, dann durchsetzen. **Ein Gate, das am ersten Tag blockiert, wird am zweiten Tag abgeschaltet.** Regulierte Compliance-Profile erzwingen `blocking` (19.6).

Im Blocking-Modus ist ein Gate-FAIL funktional gleichwertig mit einem fehlgeschlagenen Build: Beides erzeugt Feedback für den nächsten Agentenlauf. Das Gate ist damit kein Endpunkt, sondern ein **Regler** im Regelkreis aus 19.3. Die Gate-Entscheidung wird am Run persistiert und geht in Warum-Trace und Audit-Export ein — für jeden gelieferten Stand ist nachweisbar, welches Urteil ihn passieren ließ.

19.6. Governance und Nachweisführung

Dieser Abschnitt beschreibt den Teil, den die Vorversion konzeptionell nur streifte und der in regulierten Umgebungen über Einsatz oder Nicht-Einsatz entscheidet: **Wie beweist man hinterher, was ein Agent unter welchen Regeln getan hat?**

Mandanten und Rollen

Projekte, Runs und alle abgeleiteten Aggregate sind mandantengescopt; Zugriffsversuche über fremde IDs scheitern, und diese Eigenschaft ist als Test festgeschrieben. Bemerkenswert ist eine Entscheidung gegen die Konvention: `ADMIN` ist *kein* Super-Admin. Auch ein Administrator bleibt für Daten mandantengescopt; isolationsfrei sind nur Kontenverwaltung und Projektzuweisung. Eine Administrationsrolle kann Betrieb führen, ohne Einblick in fremde Projektinhalte zu haben. *Einschränkung:* Die Isolation ist an den interaktiven Pfaden verankert; die asynchrone Ausführungsschicht löst den Mandanten über einen eigenen Resolver auf.

Das fünfstufige Rollenmodell (Viewer < Developer < Maintainer < Owner < Admin) entspricht dem Least-Privilege-Modell aus Kapitel 14; freigeben darf ab Maintainer. **Segregation of Duties** ist zuschaltbar: Wer einen Run ausgelöst hat, darf ihn nicht selbst freigeben. Optionales SSO

über OIDC provisioniert neue Nutzer bewusst restriktiv als Viewer ohne Mandanten — ein SSO-Anschluss darf Zugriff nie versehentlich erweitern.

Policy-as-Code

Regeln sind kein Konfigurationszustand, sondern ein **versioniertes, signiertes Dokument**. Der kanonische Inhalt ist bewusst minimal und dadurch prüfbar: erlaubte Adapter, Gate-Modus, Pflicht-Freigabephasen, Attestierungspflicht.[6] Je Mandant existiert **genau eine aktive Version** — erzwungen über einen partiellen Unique-Index in der Datenbank plus Service-Logik. Ohne diese Invariante wäre die Frage „welche Regel galt zu diesem Zeitpunkt?“ nicht beantwortbar. Durchgesetzt wird im Orchestrator, an den zwei in 19.3 beschriebenen Zeitpunkten.

Compliance-Profile

Vier Compliance-Profile — präziser: technische Kontrollprofile mit Bezug zu ausgewählten Anforderungen — übersetzen Regulatorik in erzwingbare Policy-Vorlagen, die Brücke von der Norm zum Code:

Profil	Gate	Pflichtfreigaben	Attestierung	Regulatorischer Bezug
Baseline	advisory	—	nein	kein reguliertes Umfeld
EU AI Act	blocking	Execution	ja	VO (EU) 2024/1689, u. a. Art. 12 (Aufzeichnungspflichten), Art. 14 (menschliche Aufsicht) [14]
BAIT / MaRisk / DORA	blocking	Execution, Validation	ja	BaFin-Anforderungen an die IT; DORA (EU) 2022/2554 — IKT-Risiko, Nachweisführung [13]

[6] `PolicyInhalt` mit kanonischer Serialisierung (`kanonisch()`/`parse()`); Ed25519-Signatur über den kanonischen Text.

Profil	Gate	Pflichtfreigaben	Attestierung	Regulatorischer Bezug
BSI-Grundschutz / VS-NfD	blocking	Execution, Validation	ja	BSI IT-Grundschutz; für Verschlusssachen („Verschlusssache — Nur für den Dienstgebrauch“, VS-NfD) sind die erlaubten Adapter zusätzlich projektspezifisch auf lokale Backends zu beschränken

Das Anwenden eines Profils veröffentlicht eine neue signierte Policy-Version und erzeugt ein attestiertes Ereignis. **Ehrliche Einordnung:** Die Profile setzen die *technisch erzwingbaren* Anteile der jeweiligen Regelwerke durch — Aufzeichnung, menschliche Aufsicht, Nachweisführung, Vendor-Beschränkung. Sie ersetzen keine Rechtsberatung und decken keine organisatorischen Pflichten ab; ob ein konkreter Einsatz überhaupt in den Anwendungsbereich der zitierten Pflichten fällt — beim EU AI Act etwa die Hochrisiko-Einstufung —, entscheidet der Anwendungsfall, nicht die Plattform.

Die signierte Audit-Hashkette

Der Kern der Nachweisfähigkeit: Jedes Audit-Ereignis trägt eine lückenlose Sequenznummer, den Hash des Vorgängers, den Hash über den eigenen Inhalt, eine Ed25519-Signatur und die Schlüssel-ID.[7] Die Kettenprüfung unterscheidet drei Fehlerbilder — ein Eintrag wurde nachträglich verändert (`HASH_MISMATCH`), entfernt oder eingefügt (`CHAIN_BREAK`), oder stammt nicht vom erwarteten Schlüssel (`BAD_SIGNATURE`). Altbestand ohne Signatur wird transparent ausgewiesen, statt die Prüfung scheitern zu lassen — nachvollziehbar für Migrationen, aber das Ergebnis darf dann nicht wie eine vollständig verifizierte Kette aussehen. Konsequent im Sinne von AP-7 wären getrennte Ergebnisse (`VERIFIED`, `VERIFIED_WITH_UNSIGNED_LEGACY_PREFIX`, `UNVERIFIED_LEGACY`, `BROKEN`), von denen strikte Kontrollprofile nur `VERIFIED` akzeptieren.

Attestiert wird nicht jeder Tastendruck, sondern **jede Entscheidung, die den Handlungsspielraum des Agenten festgelegt hat**: Run-Lebenszyklus, Modellauflösung, angewendete und verweigerte Policies, angewendete Guardrails-Version, Gate-Ergebnis, Freigaben, erzeugte

[7] `AuditService.erfasse` verkettet und signiert unter einem Monitor; `AttestierungService.verifiziereKette` prüft.

und signierte Artefakte. Diese Liste ist die praktische Antwort auf die Frage, was in einem agentischen System nachweispflichtig ist.

Zwei Implementierungsdetails mit Verallgemeinerungswert: Zeitstempel werden auf Millisekunden trunkiert (gekürzt), weil die Signatur sonst nach dem Datenbank-Roundtrip nicht byte-stabil wäre — ein typischer, teuer gelernter Fallstrick beim Signieren persistenter Daten. Und der Schlüsselbetrieb ist konfigurationsgegatet: In Produktionsprofilen ist „Attestierung aktiviert, aber kein Schlüssel vorhanden“ — ein *Startfehler*; wer ohne Signatur betreiben will, muss das explizit deklarieren. Sicherheit durch bewusste Entscheidung statt durch Vergessen.

Warum-Trace und Audit-Export

Für jeden einzelnen Run beantwortet ein **Warum-Trace** die Frage „warum ist dieser Code so entstanden?“, indem er Run und Metriken (Modell, Kosten, Dauer), Plan-Herkunft, Freigabeentscheidungen samt Akteuren, verwendete Artefakte und die attestierten Ereignisse korreliert.[8] Zwei Attestierungslücken (Modellauflösung, Gate-Ergebnis) wurden eigens dafür geschlossen — der Trace erzwang rückwirkend, dass diese Ereignisse überhaupt entstehen.

Ein **Audit-Export** bündelt je Projekt oder Run ein prüffähiges Paket aus Warum-Trace, Kettenverifikation, geltendem Policy-Stand und dem öffentlichen Schlüssel — als JSON und HTML. Ein Detail mit Praxiswert: Das Bundle enthält ausschließlich Strings, Primitive und UUIDs (Zeitstempel als ISO-8601-Text). Die Serialisierung ist damit deterministisch und unabhängig von Bibliotheksversionen — ein Nachweis, der je nach Jackson-Version anders aussieht, taugt nicht als Nachweis.

Lieferkette und Wissensbestand

Der Supply-Chain-Nachweis aus Kapitel 14 ist durchgezogen: SBOM je Build (standardmäßig deaktiviert, pro Installation aktivierbar; fehlt das Scanner-Werkzeug, wird der Schritt heute transparent übersprungen — konsequent im Sinne von AP-7 wäre eine policyabhängige Behandlung: sichtbare Warnung im Baseline-Profil, `ERROR` in strikten Profilen, denn ein übersprungener Pflichtnachweis darf nicht als bestanden gelten) mit Ed25519-Signatur über den Digest, Dependency- und Lizenz-Scan als Reviewer im Gate, dazu CI-seitig Abhängigkeits-, Image- und Secret-Scans. Auch der Wissensbestand ist Governance-Objekt: Skills und Plugins liegen in einer mandantengescopten, **versionierten Bibliothek** mit typisierter Herkunft (kuratiert, übernommen, angepasst). Damit ist beantwortbar, welche Erweiterung in welcher Version zum Zeitpunkt eines Laufs im Kontext des Agenten war — die Frage, an der sonst jede Reproduzierbarkeitsdiskussion scheitert.

Die Verhaltensregeln für Agenten (`guardrails`-Slice) sind eine versionierte Ressource, deren Versions-ID ein Hash des normalisierten Inhalts ist. Bei jedem Run wird sie ins Repository projiziert: als `AGENTS.md` — dem werkzeugübergreifenden De-facto-Standard für Agenten-Anweisungen im Repository (Stand August 2026) [1] — plus eine minimale `CLAUDE.md`, die darauf verweist. **Eine Quelle, mehrere Projektionen** statt einer gepflegten Kopie je Werkzeug; welche Guardrails-Version gewirkt hat, wird attestiert.

[8] `WarumTraceService` (`provenance`-Slice, Blatt-Slice).

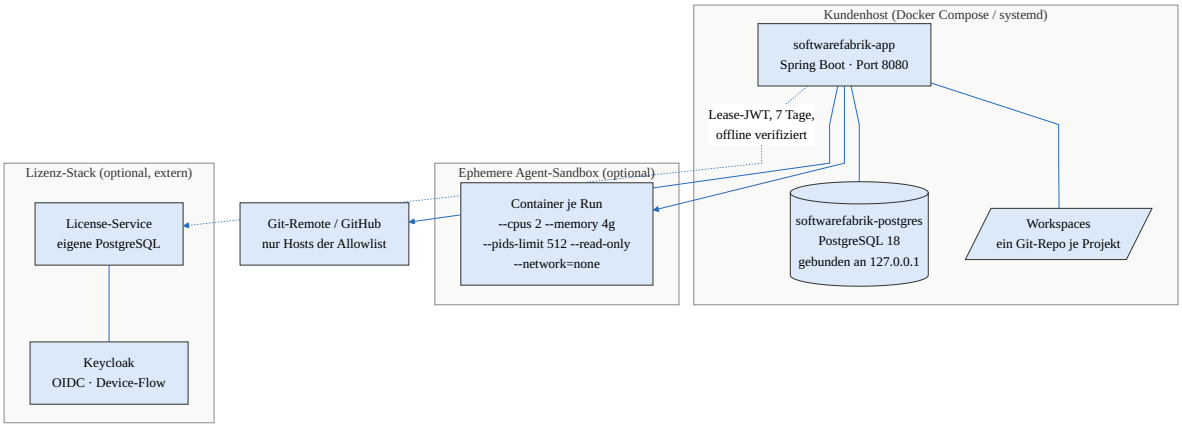

Abbildung 19.10.: Deployment-Sicht: ein Anwendungscontainer, eine Datenbank, optional getrennter Lizenz-Stack. Air-Gap-fähig

Gedächtnis ohne Vektorspeicher

Das Fünf-Schichten-Memory-Modell aus Kapitel 8 hat sich in der Praxis auf vier klar getrennte Gedächtnisformen reduziert: die versionierten Spezifikations-Artefakte (Absicht des Menschen, maschinenlesbar), das kuratierte Projektgedächtnis (`MEMORY.md`, Learnings über Läufe hinweg), der reale Code-Stand samt Git-Historie als Zustand, und das ephemere Kontextfenster des Agenten. Es gibt **keinen Vektorspeicher** — bewusst. Die Begründung ist eine Governance-Begründung: Was der Agent liest, muss ein Mensch reviewen und ein Auditor zitieren können; eine Ähnlichkeitssuche erfüllt beides nicht.

19.7. Betrieb und Souveränität

Kapitel 13 zeichnete eine Kubernetes-Deployment-Architektur. Die reale Zielgruppe — Behörden, Finanzdienstleister, Mittelstand — fragt zuerst etwas anderes: *Läuft es bei uns, ohne Cloud, ohne Rückkanal?* Die Antwort der Fabrik ist bewusst schlicht: **ein Anwendungscontainer, eine Datenbank**, dazu die Projekt-Workspaces auf dem Dateisystem und optional ephemere Agent-Container je Run. Der Lizenz-Stack (OIDC-Provider plus Lizenzdienst) ist getrennt und optional; er kann außerhalb der Unternehmensgrenze stehen oder ganz entfallen.

Die Startzeit-Härtung folgt durchgehend dem Prinzip *fail fast*: Der Compose-Stack bricht ohne Konfigurationsdatei ab statt mit Standardpasswörtern zu starten; das Container-Profil lehnt bekannte Demo-Werte und zu kurze Master-Keys hart ab; ohne gesetztes Admin-Passwort wird kein Admin angelegt; die Datenbank ist an die lokale Loopback-Adresse gebunden.

Lizenz ohne Rückkanal. Die Lizenzschicht stellt signierte Lease-Token mit sieben Tagen Gültigkeit aus, die offline verifiziert werden. Kurzzeitige Nichterreichbarkeit des Lizenzservers ist unkritisch; abgelaufenes Lease *und* nicht erreichbarer Server führen zu *fail closed*. Für die Air-Gap-Frage ist das die entscheidende Eigenschaft: Ein System, das für Verschlusssachen taugen soll, darf keine Online-Aktivierung als Betriebsvoraussetzung haben.

Damit sind alle Bausteine für den vollständig getrennten Betrieb vorhanden: lokales Modell als Adapter, netzlose Agent-Sandbox, Lizenz ohne Rückkanal, PR-Poller standardmäßig deaktiviert

(unbeaufsichtigte periodische Remote-Kontakte sind eine bewusste Betriebsentscheidung), Host-Allowlist für Remotes.

Dass das System nicht nur baubar, sondern betreibbar ist, belegt eine dauerhaft laufende öffentliche Demo-Instanz (Demo-Profil mit täglichem Datenreset). Eine Betriebserfahrung daraus gehört hierher, weil sie die Logik aus 19.5 spiegelt: Ein Ausfall der Instanz ging nicht auf die Anwendung zurück, sondern auf ein Reset-Skript, das bei vollem Dateisystem abbrach, *bevor* es die Anwendung wieder startete. Automatisierung, die bei Fehlern abbricht statt aufzuräumen, verwandelt kleine Störungen in Ausfälle — dieselbe Klasse Fehler wie ein Reviewer-Absturz, der wie ein Pass aussieht.

Eine Abgrenzung gehört an diese Stelle, weil sie leicht verwechselt wird: Die Souveränitätsaussagen dieses Abschnitts betreffen die **Plattform**, nicht die öffentliche Produktwebsite. Diese bindet seit Release 0.24.0 einen KI-Chatbot eines externen Anbieters ein, der Fragen aus den öffentlichen Website- und Dokumentationsinhalten beantwortet; das Widget lädt auf jeder Seite ein fremdes Skript und überträgt dabei die IP-Adresse, auch wenn der Chat nie geöffnet wird. Für die Air-Gap-Fähigkeit der Plattform ändert das nichts — wohl aber für die Datenschutzerklärung der Website, die Rechtsgrundlage, Auftragsverarbeitung, Speicherdauer und Serverstandort ausdrücklich als mit dem Anbieter zu vereinbaren ausweist, statt sie zu behaupten. Genau diese Trennung zwischen belegter und angenommener Aussage ist die Haltung, die auch dieses Kapitel trägt.

Erwähnenswert ist schließlich, dass die Fabrik sich selbst denselben Mechanismen unterwirft, die sie anderen Projekten auferlegt: buildbrechende Architektur-Tests und Coverage-Schwellen, Secret-Scan, SBOM, Dependency-Scan, CI als Gate vor jedem Merge. Ein Werkzeug für disziplinierte Entwicklung, das selbst undiszipliniert entwickelt würde, wäre kein glaubwürdiger Beleg.

19.8. Was die Praxis am Konzept korrigiert hat

Der Abgleich zwischen Konzept (v1.3) und Implementierung ergibt über die 23 Kapitel der Vorversion hinweg überwiegend Bestätigung, an vielen Stellen Erweiterung — und vier bewusste **Positionsverschiebungen**, die aus dem Bau eines realen Systems folgen.

(1) Ein Agent je Lauf, mehrere Prüfer — statt Hub-and-Spoke-Multi-Agent

Das Konzept (Kapitel 1, 3, 10 sowie ADR-1 in Kapitel 20) setzt auf sieben parallele Spezialagenten unter einem Orchestrator. Die Implementierung startet **einen** Agentenprozess je Run und erreicht Spezialisierung anders: über Rollen- und Teamdefinitionen im Kontext, über getrennte Plan- und Build-Runs — und vor allem über **mehrere unabhängige Prüfer** nach der Ausführung.

Die Begründung: Der Nutzen mehrerer Agenten liegt in *Perspektivenvielfalt*, und die ist beim Prüfen wertvoller als beim Erzeugen. Mehrere gleichzeitig schreibende Agenten auf einem Repository erzeugen Konfliktkosten, Zurechnungsprobleme („wer hat das geschrieben?“) und einen nicht attestierbaren Zustand. Die Fabrik verschiebt die Parallelität deshalb von der Erzeugung auf die Bewertung. Aus demselben Grund wurde die Workspace-Isolation über Git-Worktrees (ADR-2) zur **Branch-Isolation auf einem projektpersistenten Workspace**: Iteration über Läufe hinweg schlägt parallele Isolation. Der Preis ist benannt — Läufe desselben Projekts waren

zunächst nicht beliebig parallel. Die Read-only-Analyse in getrennten Worktrees kam mit 0.21.0, das parallele Schreiben mit 0.22.0 — Letzteres allerdings nicht als Rücknahme des Prinzips, sondern unter einer Besitzregel: Ein Task startet nur, wenn seine Schreibbereiche mit keinem aktiven Task kollidieren (19.10). Die Sperre gegen gleichzeitige Läufe vergleicht inzwischen den Workspace statt des Projekts.

(2) Regelkreis statt Retry — Repository-Realität als Eingabe

Das Konzept beschreibt eine Retry-Strategie (Kapitel 7.5, 9). Die Implementierung zeigt: **Ein Retry ohne neue Information wiederholt nur den Fehler.** Wertvoll wird die Wiederholung erst, wenn die Ursache zur Eingabe wird — Build-Ausgabe, Reviewer-Findings, Merge-Konfliktdateien, roter CI-Status. Besonders die letzten beiden sind die eigentliche Erkenntnis — die Beobachtung aus dem Betrieb der Fabrik: Der Agent scheitert seltener am Programmieren als an der *Realität des Repositories* — veralteter Base-Branch, fremde parallele Änderungen, fremde CI. Ein agentisches System, das diese Realität nicht als Aufgabe modelliert, endet dort, wo die interessante Arbeit anfängt.

(3) Governance ist keine Ergänzung, sondern Struktur

Im Konzept steht Compliance neben der Architektur (Kapitel 12–14). In der Implementierung ist sie in die Architektur eingewachsen: Policy-Prüfung zweimal (Anlage *und* Ausführung); Freigabe als Zustand des Laufs, nicht als Nebenprozess; jede Durchsetzung als signiertes Kettenglied; genau eine aktive Policy-Version.

Die Reifungskurve lässt sich am Migrationsverlauf ablesen: Die Migrationen V26–V33 sind ausschließlich Mandanten- und Nachweisstrukturen — und sie bestimmten *rückwirkend*, an welchen Stellen im Ablauf überhaupt Ereignisse entstehen müssen. Attestierungslücken mussten nachträglich geschlossen werden, damit der Warum-Trace vollständig ist. Das ist ein belastbares Argument gegen die verbreitete Reihenfolge „erst Funktion, dann Compliance“: **Die Nachweisstruktur lässt sich nicht sauber nachrüsten** — wer sie früher entwirft, spart die Nacharbeit.

(4) Kubernetes ist keine Voraussetzung — Souveränität schon

Das Konzept zeichnet eine Kubernetes-Deployment-Architektur (Kapitel 13). Die reale Zielgruppe fragt zuerst nach Betreibbarkeit im eigenen Haus: on-prem, notfalls air-gapped, ohne Online-Aktivierung. Die Antwort der Fabrik — ein Prozess, eine Datenbank, optionaler Lizenz-Stack, lokales Modell, netzlose Sandbox — verschiebt die Komplexität dorthin, wo sie hingehört: in die Steuerung des Nichtdeterminismus, nicht in die Betriebsinfrastruktur.

Daneben bestätigt die Implementierung zentrale Thesen des Konzepts ausdrücklich: Das Orchestrator-Prinzip trägt (eine Instanz, die Zustand besitzt und Übergänge kontrolliert, ist der Unterschied zwischen Werkzeug und Prozess — und die einzige Stelle, an der Governance ansetzen kann); Guardrails brauchen einen eigenen Schichtbegriff; Claim Verification braucht einen eigenen Prüfer; deklarative Konfiguration im Repository funktioniert — nur eben werkzeugneutral; ein geteilter Wissensstand ist notwendig und mit versionierten Dateien herstellbar. Und ADR-4 (Git als Orchestrierungsmechanismus) gilt mit einer Präzisierung: Git ist Zustandsträger — Branch,

Commit, Checkpoint, Merge, PR —, aber nicht die alleinige Wahrheit. Der autoritative Zustand liegt in der Datenbank, weil ein Auditor Fragen stellt, die `git log` nicht beantwortet: welche Policy, welches Modell, wessen Freigabe.

19.9. Grenzen und offene Punkte

Ein System, das seine offenen Punkte benennt und seine Architekturschuld zählt, ist der glaubwürdigere Beleg für die Thesen dieses Whitepapers als eines, das angeblich keine hat. Die wesentlichen offenen Punkte, Erhebungsstand 9. August 2026:

Punkt	Art
Verteilter Betrieb über mehrere Hosts (Netz-Transport, Orchestrierung); die Stufen 0 bis 4 und 6 sind umgesetzt, von Stufe 5 die Koordinationsschicht auf einem Host (Feature-Flag, Standard aus)	bewusst zurückgestellt — die Roadmap-Voraussetzung *gemessener Bedarf* ist nicht erfüllt (19.10)
Budget-Obergrenze je auslösendem Nutzer (*Seat*) — die Kostenauswertung je Seat existiert, der harte Cap wirkt je Mandant	offen
Cloud-Gateways (Bedrock/Vertex/Azure) nicht end-to-end gegen echte Credentials verifiziert	Verifikationslücke
Container-Sandbox existiert, ist aber nicht der Default; ohne Container-Runtime Rückfall auf Prozessisolation	Einschränkung
Test-Isolationsdefekt: ein Integrationstest committet unter bestimmten Bedingungen in das reale Repository	Defekt
Architektur-Altschuld: eingefrorene Modulzyklen und 13 direkte Repository-Zugriffe der Web-Schicht	dokumentierte, gezählte Schuld
Preisstatus unbekannter Modelle (heute 0 €, nötig wären `UNKNOWN`/Sicherheitsersatzwert/`FLAT_RATE`)	Budget- und Abrechnungslücke
Verifikationsstatus unsignierter Legacy-Auditdaten (undifferenziertes Ergebnis)	Nachweislücke
Policyabhängiges Verhalten bei fehlendem SBOM-Scanner (heute stets übersprungen)	Fail-Closed-Lücke

Punkt	Art
Eine Schwachstelle in einer Laufzeitabhängigkeit ohne verfügbaren Fix (seit 0.20.0, in 0.30.0 unverändert offen); das Zurückgehen auf eine ältere Version würde zwei schwerer bewertete Schwachstellen wieder öffnen	bewusst akzeptiert nach dokumentierter Risikoabwägung; betroffene Angriffsfläche, Kompensationsmaßnahmen und Ablaufdatum sind hinterlegt

Dazu vier Beobachtungen aus dem Entwicklungsverlauf (38 Releases in rund vier Monaten), die sich verallgemeinern lassen:

1. **Governance kam spät, wurde aber strukturbildend** — siehe 19.8 (3).
2. **Die härtesten Befunde lagen an den Übergängen**, nicht in den Funktionen: der Branch-Zustand nach einem abgebrochenen Merge, ein vor der Policy-Aktivierung angelegter Lauf, eine Mandanten-Policy, die eine globale unterlaufen konnte. Ab Version 0.16 dominieren Befunde aus adversarialen Multi-Agent-Reviews die Changelogs — das System wurde systematisch mit dem Ziel geprüft, seine eigenen Zusagen zu brechen. Automatismen sind einzeln korrekt und im Zusammenspiel angreifbar.
3. **Vendor-Neutralität wurde durch einen Test billig.** Solange die ArchUnit-Regeln stehen, kostet ein neuer Adapter eine Klasse und einen Test. Ohne sie wäre die Kopplung längst durch die Schichten diffundiert.
4. **Ein Gate, das sich selbst überspringt, sieht aus wie ein bestandenes Gate.** Der Abhängigkeits-Scan der eigenen CI lief über Monate ohne gültigen Zugang zur Schwachstellendatenbank: Das Zugangsgeheimnis war hinterlegt, wurde aber nur zur Vorprüfung gelesen und dem Build nie als Parameter übergeben. Weil der Job zusätzlich als nicht blockierend konfiguriert war, meldete die Oberfläche durchgehend Grün — ohne dass je ein vollständiger Scan stattgefunden hätte. Der Befund kostete keine Codezeile, sondern eine Annahme: Ein Prüfschritt muss nicht nur existieren, er muss beweisen, dass er gelaufen ist. Genau das ist der Grund für das Fail-Closed-Prinzip (AP-7, Kapitel 2) — und ein Beleg dafür, dass die Lücke im eigenen System auftrat, nicht nur im Modell. Die Konsequenz ist inzwischen Konvention: Ein mit 0.30.0 ergänzter Schemaprüfungs-Test, der ohne Container-Runtime nicht laufen kann, überspringt sich mit einer Meldung, die ausdrücklich sagt, dass Überspringen kein Bestehen ist.

Ausdrücklich **nicht** behauptet werden quantifizierte Produktivitäts- oder ROI-Zahlen aus dem Betrieb der Fabrik selbst — dafür existiert keine kontrollierte Messung. Daran ändert auch die mit 0.27.0 eingebaute und seither erweiterte Kennzahlenmessung (19.10) nichts: Sie misst Durchlaufzeiten, Quoten und Kosten der Workflows, weist aber gerade den Vergleich zur manuellen Umsetzung als Messlücke aus, weil diese Referenzgruppe im System nicht existiert. Die Modellrechnungen aus Kapitel 15 bleiben Modellrechnungen; was dieses Kapitel belegt, ist etwas anderes: dass die Referenzarchitektur als lauffähiges, betreibbares, nachweisfähiges System existiert.

Threats to Validity

Zur ehrlichen Bilanz gehört auch die Grenze der Aussagekraft dieser Fallstudie selbst: System und Whitepaper stammen vom selben Autor, die Architekturentscheidungen wurden nicht unabhängig evaluiert, und das Repository ist nicht öffentlich (interne Validität). Es existiert bisher eine Referenzimplementierung mit einer begrenzten Zahl realer Projekte; die Übertragbarkeit auf große Monorepos und verteilte Teams ist offen (externe Validität). Und die verwendeten Kennzahlen sind mit Bedacht zu lesen: Codezeilen sind kein Qualitätsmaß, Testabdeckung ist kein vollständiger Wirksamkeitsnachweis, Confidence Scores sind keine Wahrscheinlichkeiten, und bestandene Gates beweisen keine Fehlerfreiheit (Konstruktvalidität).

19.10. Vom Run zum parallelen Workflow: Ausbaustufen und Umsetzungsstand *(Roadmap)*

> **Status:** Die Roadmap-Stufen **0 bis 4 und 6 sind umgesetzt** — alles hinter einem standardmäßig deaktivierten Feature-Flag; welches Release welche Stufe brachte, zeigt die folgende Release-Verlaufstabelle. Stufe 5 — der verteilte Worker-Pool — trägt als einzige eine ausdrückliche Voraussetzung: gemessenen Bedarf. Sie ist deshalb bewusst nur zur Hälfte umgesetzt. Grundlage ist die interne Entwicklungs-Roadmap der SoftwareFabrik; Versions- und Phasenangaben sind Vorschläge, keine Zusagen.

Der Release-Verlauf 0.21.0 bis 0.30.0

Diese Tabelle ist die autoritative Zuordnung von Releases zu Roadmap-Stufen; alle anderen Stellen des Whitepapers — Management Summary, Praxis-Checks, Kennzahlen-Erhebungsvermerk (19.1) und die ADR-Statuszeilen — verweisen hierher, statt den Verlauf zu wiederholen.

Release	Stufe	Inhalt	Migrationen
0.21.0	0 + 1	Workflow-Aggregat, Task-Graph, parallele *nicht-schreibende* Analyse mit Synthese-Task und menschlicher Planfreigabe	V40–V43
0.22.0	2	Parallel *schreibende* Child Runs: Pfad-Besitzmodell, Workspace Leases, Merge Queue, Integration Gate	V44–V45
0.23.0	3	Contract Registry (eigener `contract`-Slice): versionierte, gehashte Verträge mit Stale-Erkennung	V46–V48
0.24.0	4a	Versionierte, attestierte Planänderungen; Rücksetzen nur mit neuer Eingabe	V49
0.25.0	4b	Merge Intelligence: sieben Konfliktarten, Rebase-/Revalidierungs-Pipeline, Eskalationsbericht	V50
0.26.0	5a	Koordinationsschicht: Worker-Registrierung mit Herzschlag, Anspruch vor Seitenwirkung	V51
0.27.0	6	Produktivitäts- und Qualitätsmessung; nicht Messbares als Lücke ausgewiesen statt geschätzt	keine (bewusst)

Release	Stufe	Inhalt	Migrationen
0.28.0	6	Testabdeckungsänderung (am Build-Ergebnis, in Prozentpunkten)	V52
0.29.0	6	Rollback-Erkennung als ausgewiesene Untergrenze; der Einzel-Run-Teil wirkt ohne Feature-Flag	V53
0.30.0	6	Eingriffs- und Aufwandserfassung an der Freigabeentscheidung; das Aufwandsfeld wirkt ohne Feature-Flag	V54

Stand: 9. August 2026, erhoben auf den Release-Tags. Die Details je Stufe nennt der Kasten zum Umsetzungsstand weiter unten.

Das Zielbild

Die SoftwareFabrik ist von einer Control Plane für einzelne Agentenläufe zu einer Workflow-Plattform für mehrere koordinierte Läufe weiterentwickelt worden. Der bestehende Run bleibt die atomare Ausführungseinheit — er wird nicht ersetzt, sondern zum **Child Run** eines übergeordneten **Workflows**: Ein Workflow bildet ein Feature oder Vorhaben ab und zerlegt es in **Workflow-Tasks** mit Abhängigkeiten (Task Graph). Jeder schreibende Task wird durch einen eigenen Child Run mit isoliertem Branch oder Worktree ausgeführt; Read-only-Reviewer prüfen die Child Runs parallel; ein **Merge Coordinator** führt erfolgreiche Ergebnisse kontrolliert auf einem Integrationsbranch zusammen, und ein abschließendes **Integration Gate** validiert den Gesamtstand. Die Leitregel:

> Parallelität findet zwischen isolierten Tasks und Runs statt. Innerhalb eines Workspace existiert genau ein schreibender Agent.

Die tragenden Prinzipien

1. **Single Writer per Workspace.** Pro Branch, Worktree oder Arbeitskopie schreibt zu einem Zeitpunkt genau ein Agent — das Prinzip, das sich im Einzel-Run bewährt hat, wird auf jede Parallelitätseinheit übertragen.
2. **Parallelism by Dependency.** Parallelität wird aus Task-Abhängigkeiten, Schreibbereichen (Owned/Read-only/Protected Paths), Verträgen und Risikopolicies abgeleitet — nicht aus festen Agentenrollen.
3. **Contracts before Parallel Implementation.** Voneinander abhängige Komponenten werden erst parallel implementiert, wenn gemeinsame Verträge versioniert vorliegen (OpenAPI, AsyncAPI, Interfaces, Domain Events, Schemas); jeder Child Run attestiert, gegen welche Vertragsversion er gearbeitet hat, und Vertragsänderungen setzen betroffene Tasks auf Neuplanung.
4. **Independent Verification, zweistufig.** Jeder Child Run durchläuft ein lokales Task-Gate (Syntax, Style, Security, Domain, Tests, Claim Verification); der zusammengeführte Stand

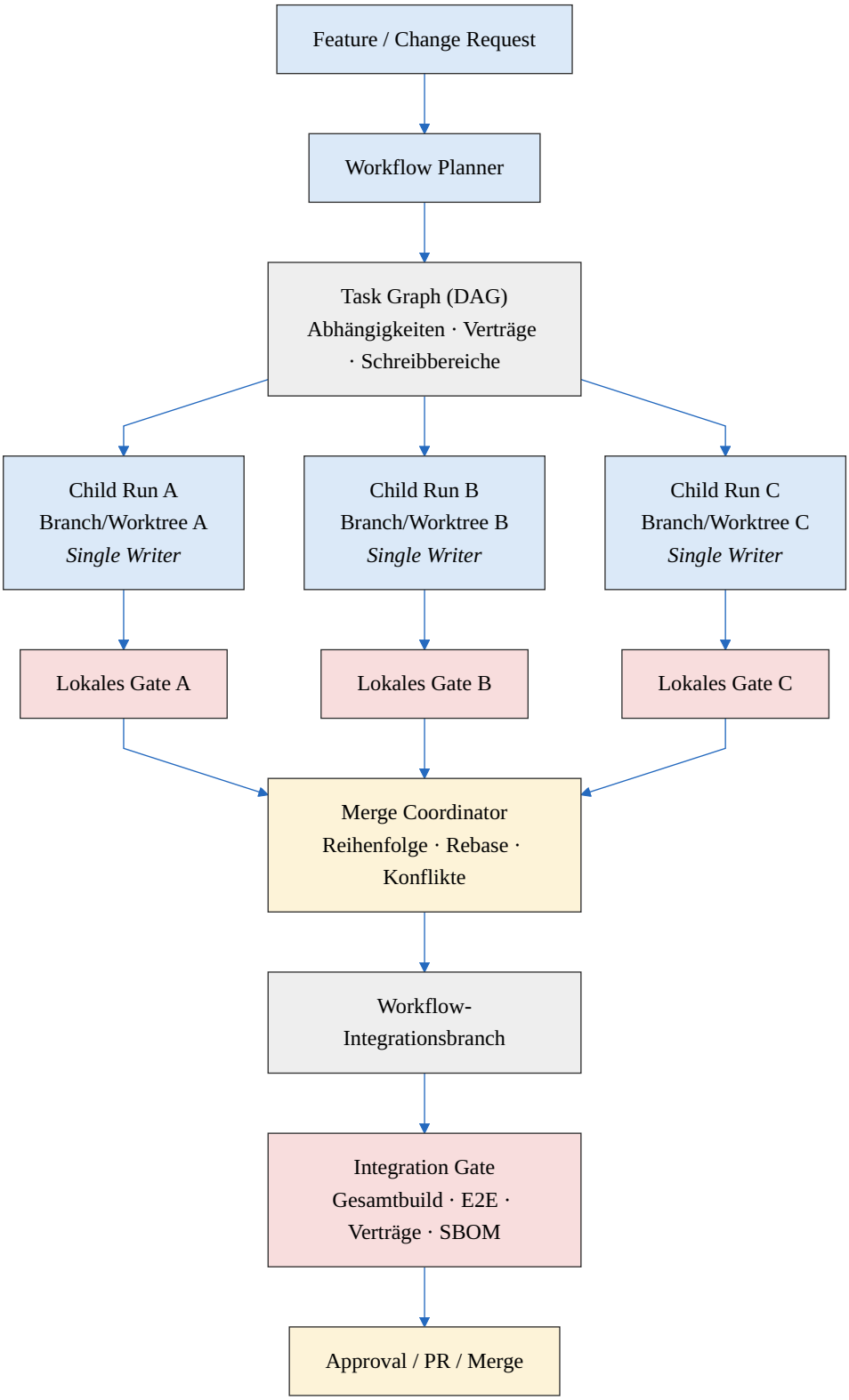

Abbildung 19.11.: Architektur der parallelen Agenten-Workflows: isolierte Child Runs unter einem Parent Workflow, zusammengeführt über Merge Coordinator und Integration Gate. Bis Release 0.30.0 umgesetzt, hinter deaktiviertem Feature-Flag

durchläuft zusätzlich das Integration Gate (Gesamtbuild, Regression, End-to-End, Verträge, Migrationen, Gesamt-SBOM).

5. **Workspace Leases.** Zeitlich begrenzte, technisch erzwungene Reservierungen von Pfaden, Modulen und exklusiven Ressourcen (etwa Build-Konfiguration, Datenbankmigrationen, gemeinsame API-Spezifikationen) verhindern konkurrierende Schreibzugriffe, bevor sie entstehen.
6. **Rule Loop statt Retry und Fail Closed** gelten unverändert — auch auf Workflow-Ebene: Replanning nur bei neuen Informationen; ein ausgefallener Pflicht-Reviewer, eine nicht prüfbare Policy oder eine fehlgeschlagene Attestierung gilt niemals als Erfolg.

Die Ausbaustufen

Stufe	Inhalt	Risiko
0	Architektur- und Datenmodellvorbereitung: Parent-Child-Referenzen, Workflow-Ereignisse in Audit und Warum-Trace, Feature-Flag; bestehende Einzel-Runs bleiben unverändert lauffähig	niedrig — **umgesetzt**
1	Parallele Read-only-Analyse: mehrere Analyse-Runs (Requirements, Architektur, Security, Testplanung) parallel, Synthese-Task, menschliche Planfreigabe — Planung bleibt ohne Änderungsrisiko, weil keine Schreibrechte	niedrig — **umgesetzt**
2	Parallele Child Runs für unabhängige Module: Branch/Worktree je Task, Workspace Leases, Merge Queue, lokales Gate je Child Run, Integration Gate	mittel — **umgesetzt**
3	Vertragsbasierte Parallelisierung: Contract Registry, Content-Hash je Vertrag, automatische Stale-Erkennung, Consumer-/Provider-Vertragstests	mittel–hoch — **umgesetzt** (ohne Consumer-/Provider-Vertragstests)

Stufe	Inhalt	Risiko
4	Dynamisches Replanning und Merge Intelligence: versionierte Planänderungen, Konfliktklassifikation, Rebase-/Revalidierungs-Pipeline, Eskalation mit vollständigem Kontext	hoch — **umgesetzt**
5	Distributed Worker Pool (nur bei gemessenem Bedarf): persistente Task-Queue, Worker-Leasing, horizontale Skalierung — Single-Host- und Air-Gap-Betrieb bleiben erhalten	optional — Koordinationsschicht **umgesetzt**, Netz-Transport bewusst zurückgestellt
6	Produktivitäts- und Qualitätsmessung: Durchlauf-, Kosten- und Konfliktkennzahlen, abgeleitet aus ohnehin entstehenden Daten; nicht Messbares wird als Lücke ausgewiesen statt geschätzt	niedrig — **umgesetzt**

Umsetzungsstand der Stufen 0 bis 6 (Releases 0.21.0 bis 0.30.0, Stand 9. August 2026): Die Stufen 0 bis 4 und 6 sind vollständig umgesetzt, von Stufe 5 die Koordinationsschicht; das Feature-Flag steht standardmäßig auf **aus**, ohne es verhält sich die Workflow-Ebene unverändert. Zwei Ausnahmen wirken auch ohne Flag, weil sie am Run-Aggregat beziehungsweise an der Freigabeentscheidung sitzen: der Einzel-Run-Teil der Rollback-Erkennung aus 0.29.0 und das freiwillige Aufwandsfeld aus 0.30.0 (siehe unten).

Stufe 0 trägt die beiden Architekturentscheidungen — hierarchische Orchestrierung mit der strikten Richtung Workflow führt zu Run, niemals umgekehrt, und Single Writer per Workspace —, einen eigenen `workflow`-Slice, die Zuordnung eines Runs zu Workflow und Workflow-Task sowie die Nachweisanbindung: acht Workflow-Ereignisse in der Audit-Kette (von der Anlage über Plan-Einreichung und Planfreigabe bis zu Start und Ende jeder Task) und der Workflow-Kontext im Warum-Trace jedes Child Runs. Die Zuordnung ist einmalig — ein zweiter Aufruf wird abgewiesen, damit die historische Zurechenbarkeit erhalten bleibt.

Stufe 1 führt die parallele Analyse aus: ein Workflow-Aggregat mit elf Zuständen, Tasks mit zwölf Zuständen und einem Abhängigkeitsgraphen, höchstens drei gleichzei-

tige Child Runs, jeder in einem eigenen Worktree. In dieser Stufe wurden ausschließlich **nicht-schreibende** Capabilities ausgeführt; das Fähigkeitsmodell kennt 22 Capabilities, davon acht schreibende, die erst Stufe 2 freigeschaltet hat. Die menschliche Planfreigabe ist ein eigener Zustand, kein Nebenkanal, und das Workflow-Budget greift *vor* dem Start der nächsten Task.

Bemerkenswert ist der Zuschnitt des **Synthese-Schritts**: Er hängt automatisch an allen Tasks, baut seinen Auftrag aus deren Ergebnissen und hält fest, welche Eingaben eingeflossen sind — die Referenzen sind damit belegt, nicht behauptet. Sein Auftrag fordert Widersprüche zwischen den Analysen ausdrücklich ein und verbietet, sie zu glätten. Das ist dieselbe Haltung wie bei der Claim Verification im Einzel-Run (19.5): Der Wert mehrerer Perspektiven entsteht dort, wo sie sich widersprechen.

Stufe 2 (Release 0.22.0) erlaubt Tasks, Produktivcode zu ändern. Damit das zurechenbar bleibt, kommen zwei Mechanismen hinzu — eine Besitzregel **vor** der Ausführung und eine kontrollierte Zusammenführung **danach**.

Jeder Task erklärt seine Pfadbereiche in drei Arten: `OWNED` (darf schreiben), `READ_ONLY` (darf lesen) und `PROTECTED` (Schutzzone). Überschneiden sie sich mit denen eines aktiven Tasks, startet er gar nicht erst — der Konflikt wird verhindert, nicht später erkannt. Verglichen wird auf Segmentgrenzen, damit `src/main` und `src/mainx` nicht fälschlich kollidieren. Die Leases tragen eine Ablauffrist und einen Herzschlag: Ein abgestürzter Agent legt den Workflow nicht dauerhaft lahm. Build-Dateien und das Migrationsverzeichnis sind **immer** exklusiv, auch wenn ein Task sie nicht anmeldet — die Erfahrung aus 19.8 (2), dass die härtesten Befunde an den Übergängen liegen, ist hier zur Regel geworden.

Nach erfolgreicher Ausführung führt eine **Merge Queue** die Child-Branches *sequenziell* und in *Planreihenfolge* in einen Integrationsbranch — nicht in der Reihenfolge, in der sie zufällig fertig wurden, sonst hinge das Ergebnis an Laufzeiten. Erst der zusammengeführte Gesamtstand durchläuft das **Integration Gate**: Das lokale Gate je Child Run prüft eine Änderung *für sich*, das Integration Gate prüft, ob sie *zusammen* funktionieren. Nur über dieses Urteil erreicht ein Workflow den Zustand `COMPLETED`.

Zwei Entscheidungen sind bemerkenswert, weil sie dem Fail-Closed-Prinzip folgen statt der Bequemlichkeit. Ein **Merge-Konflikt hält an, statt zu scheitern**: Der Workflow geht in die Freigabe und zeigt die Konfliktdateien; der Mensch entscheidet zwischen erneutem Versuch (begrenzt auf zwei) und dem Verwerfen des Branches. Ein verworfener Branch macht das Integration Gate `FAILED` — ein Workflow, dessen Arbeit teilweise liegen blieb, gilt nicht als erfolgreich. Und ein **Kaskaden- Abbruch** stoppt bei einem gescheiterten Task alle, die mittelbar auf ihm aufbauen, und verwirft offene Merge-Einträge; ohne ihn blieben Nachfolger blockiert und der Workflow käme nie zu einem Ende.

Stufe 3 (Release 0.23.0) gibt den Verträgen einen Ort. Bis dahin konnten zwei Tasks gegen dieselbe Schnittstelle arbeiten, ohne dass festgehalten war, gegen **welchen** Stand. Ein eigener Blatt-Slice `contract` verwaltet unveränderliche Fassungen mit

Nummer und Content-Hash über den normalisierten Inhalt; sechs Vertragsarten sind vorgesehen, von OpenAPI und AsyncAPI über JSON-Schema und Java-Interfaces bis zu Domain Events und Akzeptanzkriterien. Drei Entscheidungen tragen die Konstruktion. Die **Normalisierung ist bewusst minimal** — nur Zeilenenden und abschließende Leerzeichen; eine Umformatierung gilt als Änderung, denn ein falsches „veraltet“ kostet eine erneute Prüfung, ein übersehenes einen stillen Vertragsbruch. Die **Bindung geschieht beim Start** des Child Runs, nicht bei der Planung: Im Plan steht nur der Name, denn alles andere wäre eine Wette darauf, dass sich bis zum Start nichts ändert. Und die **Stale-Erkennung** setzt jeden gebundenen, nicht abgeschlossenen Task synchron in der Veröffentlichungstransaktion auf Neuplanung — es gibt kein Fenster, in dem ein Task noch als aktuell gilt. Blockiert wird zweimal, vor dem Einreihen in die Merge Queue *und* vor dem Merge, weil sich ein Vertrag ändern kann, während der Eintrag wartet.

Stufe 4a (Release 0.24.0) macht Planänderungen nachweisbar. Bisher sagte die Planversion, *welcher* Plan gilt — nicht, was sich geändert hat und warum; genau das wird aber gefragt, meist Wochen später. Jede Änderung liegt nun unveränderlich als Revision vor, mit Grund, Pflichtbeschreibung, Auslöser und betroffenen Tasks, attestiert über die Audit-Hashkette. Bemerkenswert ist, wie hier das Rule-Loop-Prinzip (Regelkreis statt blindem Retry, Kapitel 2) hart erzwungen wird: Ein Task lässt sich nur zurücksetzen, wenn der Grund eine **neue Eingabe** trägt. Von den acht möglichen Gründen zählen sechs als neue Information — Vertragsänderung, Merge-Konflikt, Build-Fehler, Reviewer-Befund, gescheiterter Task, menschliche Anweisung. Ein anderer Zuschnitt und eine neue Reihenfolge zählen ausdrücklich **nicht**: Sie ordnen Arbeit um, liefern dem Agenten aber nichts, was er beim letzten Versuch nicht schon hatte. Damit ist der blinde Retry, den 19.8 (2) als Irrweg beschreibt, technisch ausgeschlossen statt nur empfohlen.

Beim Umschneiden des Plans zeigt sich dieselbe Sorgfalt: Teilt man einen Task, erben die Teile Abhängigkeiten und Capability, aber *nicht* die Schreibbereiche — sonst könnten sie nie parallel laufen und das Aufteilen wäre sinnlos. Führt man Tasks zusammen, erbt der neue umgekehrt die Vereinigung, denn wer alles hält, was vorher verteilt war, erzeugt keinen Besitzkonflikt. Ein *laufender* Task lässt sich gar nicht umschneiden: Den Zuschnitt unter einem arbeitenden Agenten zu ändern, machte das Ergebnis niemandem mehr zurechenbar.

Stufe 4b (Release 0.25.0) macht aus „Merge-Konflikt“ eine Diagnose. Bis dahin war das eine Sammelkategorie: Zwei konkurrierende Migrationsnummern, eine doppelt hinzugefügte Abhängigkeit und ein echter logischer Widerspruch sehen für Git gleich aus, verlangen aber völlig verschiedene Reaktionen. Sieben Konfliktarten werden nun unterschieden, geprüft von der teuersten zur harmlosesten — eine Migrationskollision, die nebenbei einen Formatierungskonflikt enthält, bleibt eine Migrationskollision; die umgekehrte Einordnung lüde zum Überschreiben ein. Die Analyse läuft nebenwirkungsfrei im Objektspeicher, ohne Arbeitsbaum und Index anzufassen, damit der Integrations-Worktree währenddessen einen definierten Stand behält.

Zwei Details passen zur Argumentation dieses Kapitels. Ein **Rebase wird vor dem Merge versucht, nicht statt seiner** — und bei Migrationskollisionen sowie inhaltlichen Widersprüchen ausdrücklich nicht, obwohl er technisch möglich wäre: Er verschöbe diese Konflikte nur, und ein Versuch ohne neue Information ist genau der blinde Retry, den das Rule-Loop-Prinzip verbietet. Und der **Eskalationsbericht** nennt Branch, Task, Capability, Konfliktart, Dateien, bisherige Versuche, das Ergebnis eines etwaigen Rebase, die beteiligten Vertragsfassungen und eine Empfehlung — wer eskaliert wird, hat den Vorgang nicht verfolgt, und „Konflikt in drei Dateien“ zwänge ihn, den halben Zustand selbst zu rekonstruieren.

Stufe 5a (Release 0.26.0) ist der interessanteste Fall, weil hier eine Voraussetzung ernst genommen wurde. Die Stufe gilt laut Roadmap nur bei **gemessenem Bedarf** — und der besteht für verteilten Betrieb nicht: Die Workflow-Ebene läuft hinter einem standardmäßig deaktivierten Flag, und ein Pool über mehrere Hosts brächte Zugangsdaten und Workspace-Zugriff auf weitere Maschinen, also Angriffsfläche für eine Last, die es nicht gibt. Der Netz-Transport ist deshalb bewusst zurückgestellt. Dieselbe Maschinerie schloss aber auf *einem* Host eine reale Lücke: Tasks starteten nur auf Knopfdruck, ein durch seinen Vorgänger frei gewordener Nachfolger blieb liegen; nach einem Neustart nahm niemand laufende Arbeit wieder auf; und der Reservierungszustand `CLAIMED` stand seit Stufe 1 im Zustandsmodell, ohne je gesetzt zu werden. Worker-Registrierung mit Ablaufzeit und Herzschlag schließt das — nach demselben Muster wie die Workspace Leases, weil ein abgestürzter Prozess nichts mehr freigeben kann. Ein geordnet abgemeldeter Worker gilt ausdrücklich nicht als verwaist, sonst liefe bei jedem Herunterfahren eine Aufräumaktion.

Drei Entscheidungen dieser Stufe tragen die Handschrift des Kapitels. **Anspruch vor Seitenwirkung:** Ein Worker beansprucht einen Task, bevor Worktree und Child Run entstehen — zwei Prozesse können nicht beide gewinnen; das ist die technische Form von „ein Task wird höchstens einmal aktiv geschrieben“. **Kein stiller Erfolg beim Verfall:** Verfällt der Anspruch eines toten Workers auf einen Task, der noch nicht lief, geht der Task zurück nach `READY` — es ist nichts geschehen. Lief er bereits, geht er nach `FAILED`, weil der Workspace in unbekanntem Zustand sein kann; ihn still zu wiederholen wäre genau der blinde Retry, den das Rule-Loop-Prinzip verbietet — der Weg zurück führt über das begründungspflichtige Replanning der Stufe 4a. Und der **automatische Versand** freigewordener Nachfolger sitzt hinter einem eigenen Schalter mit Standardwert aus: Automatisches Starten von Läufen kostet Tokens — das schaltet man bewusst ein.

Eine Nebenwirkung vom Beginn dieser Entwicklungslinie (Release 0.21.0) verdient hier noch Erwähnung, weil sie ein Muster dieses Kapitels bestätigt: Die Sperre gegen gleichzeitige Läufe war projektweit formuliert, begründete sich in ihrem eigenen Kommentar aber mit dem *geteilten Workspace*. Sie vergleicht seither den effektiven Workspace — getrennte Worktrees dürfen parallel laufen, dasselbe Verzeichnis weiterhin nicht. Die Regel wurde dadurch **präziser, nicht schwächer**.

Stufe 6 (Release 0.27.0) baut die begleitende Produktivitäts- und Qualitätsmessung

ein — als 30. Fachlichkeit, bewusst **ohne eigenes Schema**: Alle Werte werden aus Daten abgeleitet, die ohnehin entstehen, denn eine Kennzahl, die in einer eigenen Tabelle mitgeführt wird, weicht früher oder später von dem ab, was sie beschreiben soll. Messbar sind Time to Accepted Merge, Erstdurchlauf-Quote, Korrekturschleifen, Planänderungen, Kosten je Workflow und je Child Run, die Merge-Konfliktquote samt Verteilung nach Konfliktart (aus Stufe 4b) und die Freigabe-Wartezeit. Vier Messgrößen der Roadmap wies 0.27.0 ausdrücklich als Lücke mit Begründung aus, statt sie zu schätzen; drei davon bestehen fort — eine seit 0.30.0 zerlegt, eine seit 0.29.0 halbiert, eine unverändert: die menschliche aktive Arbeitszeit (die Plattform sieht, wie lange etwas *wartete*, nicht, ob jemand daran *arbeitete* — beides gleichzusetzen wäre die verlockendste und falscheste Kennzahl; siehe unten), entkommene Defekte (siehe unten) und der Vergleich zur manuellen Umsetzung (diese Referenzgruppe existiert im System nicht). Eine Kennzahl, die anders heißt als das, was sie misst, ist schlimmer als eine fehlende — sie wird geglaubt. Und eine Quote ohne Nenner ist *unbekannt*, nicht null: Ein Workflow ohne einen einzigen Merge hat keine Konfliktquote von 0 % — er hat gar keine; die Oberfläche zeigt dafür einen Strich.

Die vierte Lücke aus 0.27.0 — die Testabdeckungsänderung — schloss Release 0.28.0: Der Abdeckungsbericht liegt ohnehin im Workspace, er wurde nur nie gelesen. Die Konstruktion trägt dieselbe Haltung wie der Rest der Stufe. Gelesen wird in fester Formatreihenfolge (JaCoCo, Cobertura, Istanbul), damit die Kennzahl nicht davon abhängt, welche Datei das Dateisystem zuerst nennt. Die Abdeckungsspalten am Build-Ergebnis (Migration V52) sind bewusst **nullable** — ein Projekt ohne Abdeckungsbericht hat nicht null Prozent Abdeckung, man weiß es schlicht nicht. Die Änderung wird in **Prozentpunkten** ausgewiesen, nicht in Prozent, und erscheint erst ab der zweiten Messung — eine Änderung gegenüber nichts ist keine Änderung. Und der Bericht wird bewusst **ohne XML-Parser** gelesen: Er stammt aus agentengeneriertem Code und ist damit nicht vertrauenswürdig; ein vollwertiger Parser wäre eine XXE-Angriffsfläche direkt im Eingabepfad.

Release 0.29.0 halbierte die Lücke „entkommene Defekte und Rollbacks“. Für die Rollback-Hälfte stimmte die Begründung — die Plattform ende beim Merge — nämlich nicht: Ein `git revert` steht in derselben Historie, die die Fabrik beim nächsten Lauf ohnehin synchronisiert; was fehlte, war der **Anker**, der festgehaltene Merge-Commit, auf den sich ein Revert beziehen kann (Migration V53, an Merge-Queue und Run). Drei Entscheidungen halten die Kennzahl ehrlich: Gemessen wird der **Revert, nicht der Defekt** — ein Revert-Commit ist eine Tatsache in der Historie, „ein späterer Commit berührte dieselbe Datei“ wäre eine Vermutung und ebenso gut das nächste Feature. Die **Quote ist eine Untergrenze und heißt auch so** — wer eine Änderung von Hand rückwärts anwendet und ohne Revert-Vermerk committet, taucht nicht auf. Und **nur verankerte Auslieferungen stehen im Nenner** — einen Merge ohne festgehaltenen Commit als „nicht zurückgenommen“ zu zählen, wäre ein geschönter Nenner. Erkannte Rollbacks werden attestiert. Aufschlussreich ist der Zuschnitt: zwei getrennte, dünne Anwender für Workflow und Einzel-Run, weil der `run`-Slice nach

der strikten Richtung aus ADR-5 nichts von der Workflow-Ebene wissen darf — und weil der Einzel-Run der Normalfall ist, wirkt dieser Teil der Messung auch ohne das Workflow-Feature-Flag. Die verbleibende Messlücke heißt nur noch „entkommene Defekte“; sauber schließen ließe sie sich erst über eine Ticketsystem-Anbindung, die es nicht gibt.

Release 0.30.0 nahm sich die verbliebene Arbeitszeit-Lücke vor — und **zerlegte sie, statt sie zu schließen**, denn von den drei denkbaren Wegen ist keiner eine Messung: Wartezeit als Arbeitszeit auszugeben wäre falsch beschriftet, Menschen zu instrumentieren wäre Überwachung (und ein offener Browser-Tab ist trotzdem keine Arbeit), und Fragen ist eine Selbstauskunft. Getrennt wurde, was sich trennen lässt. **Messbar** sind seither die *Eingriffe* — wie oft ein Mensch entscheiden musste — und die *Entscheiderzahl*, beide aus der Freigabeentscheidung abgeleitet; das beantwortet die eigentlich interessante Frage, wie viel menschliche Beteiligung ein Vorhaben braucht, ohne vorzugeben, Arbeitszeit zu messen. **Erfragt, nicht gemessen** ist der Aufwand: ein freiwilliges Feld an der Freigabe (Migration V54), bewusst nullable — leer heißt „nicht erfasst“, nicht „null Minuten“ — und mit einer Plausibilitätsgrenze von einer Woche gegen Tippfehler; die Selbstauskunft wird strikt getrennt ausgewiesen und nie mit abgeleiteten Werten verrechnet. Das Feld steht an jeder Freigabeentscheidung eines Laufs — auch der des Einzel-Runs, es wirkt also wie der Einzel-Run-Teil der Rollback-Erkennung ohne Workflow-Flag. Der Ehrlichkeitsanker ist die **Aufwandsabdeckung**: Neben der Summe steht immer, aus wie vielen Entscheidungen sie stammt — „45 min (aus 3/12)“ macht sichtbar, dass neun Entscheidungen keine Angabe tragen; ohne diese Quote würde die Summe als Gesamtaufwand gelesen, also genau die Verwechslung, gegen die die Lücke überhaupt formuliert wurde. Bewusst **nicht** gebaut: Interaktions-Telemetrie und Leerlauf-Erkennung — das ergäbe eine Zahl, aber um den Preis, Menschen zu vermessen, und sie wäre trotzdem geraten. Die Arbeitszeit selbst bleibt damit ausgewiesene Messlücke.

Die begleitende **Produktivitäts- und Qualitätsmessung** ist mit 0.27.0 eingebaut und mit 0.28.0 bis 0.30.0 um Testabdeckungsänderung, Rollback-Erkennung sowie Eingriffs- und Aufwandserfassung ergänzt (Stufe 6, siehe oben) — mit ausgewiesenen Lücken: Gemessen wird, was aus den Daten der Plattform ableitbar ist; die aktive menschliche Arbeitszeit, entkommene Defekte und der Vergleich zur manuellen Umsetzung sind als Messlücken benannt statt geschätzt. Erst eine kontrollierte Vergleichsmessung kann die Wirtschaftlichkeitshypothesen aus Kapitel 15 in belegte Aussagen verwandeln (vgl. Kapitel 15.6).

Ausdrückliche **Nicht-Ziele** der ersten Ausbaustufen: freie Agent-zu-Agent-Chats, unbegrenzte Parallelität, automatische Architekturentscheidungen ohne Freigabe, autonomes Produktionsdeployment, Kubernetes als Pflicht und empirisch unbelegte Produktivitätsversprechen.

Die neue Position

Damit lässt sich die Entwicklungslinie dieses Whitepapers in einem Absatz zusammenfassen:

Die Implementierung hat die ursprüngliche Vorstellung mehrerer gleichzeitig schreibender Rollenagenten zunächst korrigiert (19.8). Die Weiterentwicklung verwirft diese Praxiserkenntnis nicht, sondern generalisiert sie: Mehrere Agenten dürfen parallel arbeiten, sofern Schreibbereiche, Verträge und Zustände isoliert und durch einen übergeordneten Workflow kontrolliert werden. Der Weg dorthin — die Release-Verlaufstabelle am Anfang von 19.10 — ist selbst eine Aussage, denn seine Reihenfolge folgt dem Risiko: erst die Koordination beweisen, dann die Schreibrechte verteilen, dann die Verträge festhalten, dann das Umplanen nachweisbar machen, dann messen — und den verteilten Betrieb erst, wenn der Bedarf gemessen ist. Von einem kontrollierten Agentenlauf zu parallelen Agenten-Workflows — ohne das Single-Writer-, Governance- und Nachweisprinzip aufzugeben.

Teil VII.

Architekturentscheidungen & Referenz

20. 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.

Die ADRs 1–4 stammen aus Version 1.3 (März 2026) und sind bewusst im Original erhalten — sie dokumentieren, was damals entschieden wurde. Der **Status-Block** direkt unter jeder Überschrift zeigt, was seither aus der Entscheidung geworden ist: Die Referenzimplementierung (Kapitel 19) hat mehrere davon korrigiert, die Zielarchitektur (19.10) einige weiterentwickelt. Ersetzte Entscheidungen stehen als eigene ADRs 5–8 am Ende dieses Kapitels. Die Statusangaben trennen dabei zwei Dinge, die oft vermischt werden: der **Entscheidungsstatus** sagt, ob eine Architekturentscheidung gilt; der **Implementierungsstatus** sagt, ob sie im Produkt bereits umgesetzt ist. Eine angenommene Entscheidung kann also durchaus noch der Umsetzung harren.

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

> **Entscheidungsstatus:** *Superseded* (ersetzt) — Entscheidung vom März 2026 (v1.3), ersetzt im August 2026 (v2.1) durch **ADR-5: Hierarchische Orchestrierung mit Parent Workflow und Child Runs**. Die Praxis hat gezeigt, dass mehrere gleichzeitig schreibende Rollenagenten Konfliktkosten, unklare Zurechnung und nicht attestierbare Zwischenstände erzeugen (19.8). Der folgende Originaltext bleibt zur Dokumentation der ursprünglichen Entscheidungsgrundlage erhalten. Die darin enthaltenen quantitativen und qualitativen Wirkannahmen (Token-Ersparnisse, Qualitäts- und Latenzeffekte, Kontext-Overhead) dokumentieren den damaligen Entscheidungsstand; sie wurden nicht kontrolliert gemessen und gelten nicht als Ergebnisse dieser Arbeit. Das gilt sinngemäß auch für die historischen ADRs 2 bis 4.

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)
Orchestrator	Task-Dekomposition, Agent-Zuweisung, Ergebnis-Aggregation, Conflict Resolution	Opus
Architecture Agent	ADR-Erstellung, Architektur-Validierung, Bounded-Context-Prüfung	Opus
Planning Agent	Task-Dekomposition, Abhängigkeitsanalyse, Schätzung	Sonnet
Requirements Agent	Anforderungsanalyse, Akzeptanzkriterien, User-Story-Verfeinerung	Sonnet
Development Agent	Code-Generierung, Refactoring, Implementierung	Sonnet / Opus (je nach Komplexität)
Testing Agent	Testgenerierung, Coverage-Analyse, Mutation Testing	Sonnet
Review Agent	Code-Review, Architektur-Compliance, Security-Check	Opus
Deployment Agent	CI/CD-Konfiguration, Infrastructure as Code, Rollout-Strategien	Sonnet / Haiku

Die Werkzeug- und Kontextzuordnung derselben Rollen:

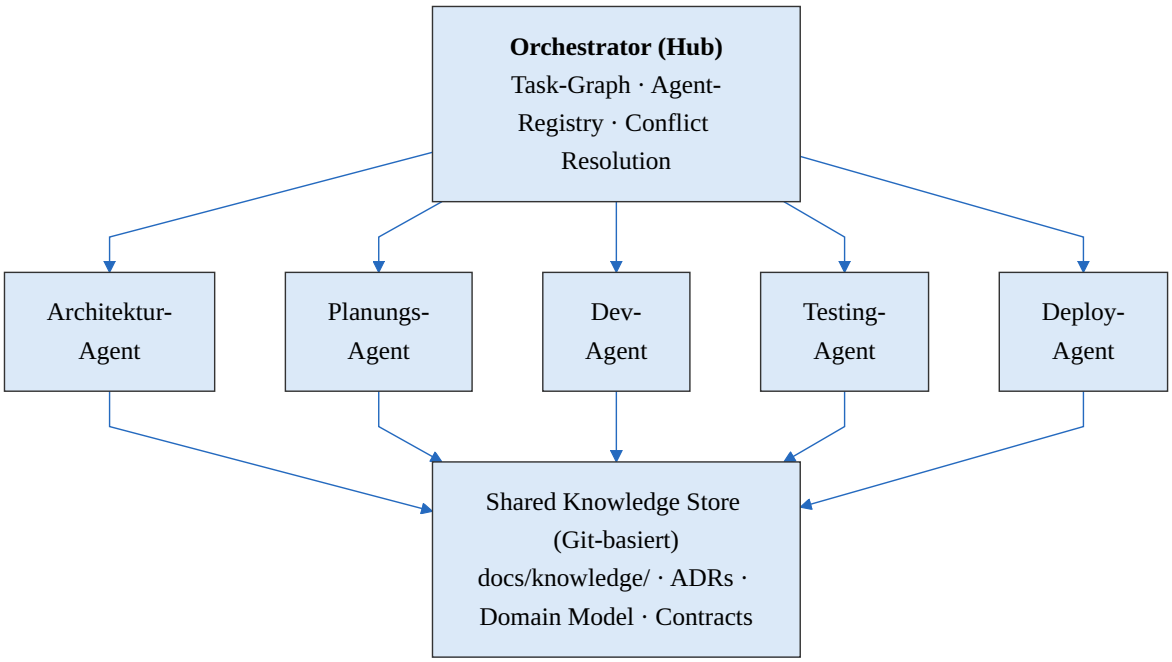

Abbildung 20.1.: Hub-and-Spoke-Topologie

Agent	Tool-Zugriffe	Kontext-Fokus
Orchestrator	Alle Agenten, Shared Knowledge Store, Git	Gesamtarchitektur, Task-Graph, Agent-Status
Architecture Agent	Dateisystem (read), Shared Knowledge Store	ADRs, Architekturregeln, DDD-Modell
Planning Agent	Shared Knowledge Store, Issue Tracker	Anforderungen, Task-Graph, Velocity-Daten
Requirements Agent	Shared Knowledge Store, Dokumentation	Fachdomäne, bestehende Requirements
Development Agent	Dateisystem (read/write), Build-Tools, Terminal	Service-Code, Dependencies, API-Contracts
Testing Agent	Dateisystem, Test-Runner, Coverage-Tools	Test-Code, Fixtures, Coverage-Reports
Review Agent	Dateisystem (read), Guardrails-Pipeline	Diff, ADRs, Security-Policies, Style-Guides
Deployment Agent	Dateisystem, Docker, K8s-Config	Deployment-Manifeste, Helm Charts, Pipelines

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

Relativfaktoren und die folgende Ersparnisangabe beziehen sich auf den Preisstand der v1.3 (März 2026); mit dem Preisstand 08/2026 aus 15.1 lägen die Faktoren bei rund 0,4x (Sonnet) bzw. 0,2x (Haiku).

Diese Differenzierung senkt die Token-Kosten modellhaft um 60–70 % gegenüber einer reinen Nutzung des stärksten Modells bei angenommen vergleichbarer Ergebnisqualität (Annahme auf v1.3-Preisstand, nicht gemessen).

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: modellhaft 60–70 % niedrigere Token-Kosten gegenüber reiner Nutzung des stärksten Modells (Annahme auf v1.3-Preisstand, nicht gemessen)
- Parallelisierung unabhängiger Tasks: angenommene Reduktion der Wall-Clock-Time um Faktor 2–4 bei Multi-Service-Features (nicht gemessen; Integrationskosten nicht eingerechnet)
- 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

> **Praxis-Check SoftwareFabrik (abweichend):** Die Implementierung startet einen Agentenprozess je Lauf und erreicht Spezialisierung über Rollen- und Teamdefinitionen im Kontext sowie über mehrere unabhängige Prüfer nach der Ausführung — Perspektivenvielfalt ist beim Prüfen wertvoller als beim Erzeugen, und mehrere gleichzeitig schreibende Agenten erzeugen Konfliktkosten und einen nicht attestierbaren Zustand (19.8). *Status (v2.2): durch die Praxis korrigiert — und in ADR-5 generalisiert: Die Workflow-Ebene erlaubt seit Release 0.22.0 auch parallel schreibende Child Runs — unter Single-Writer-Isolation je Workspace und einer Besitzregel, die überschneidende Schreibbereiche vor dem Start blockiert (Feature-Flag, Standard aus; 19.10).*

ADR-2: Workspace Isolation via Git Worktrees

> **Entscheidungsstatus:** *Amended* (teilweise ersetzt und weiterentwickelt) — Entscheidung vom März 2026 (v1.3), fortgeschrieben im August 2026 (v2.1). Die aktuelle Referenzimplementierung verwendet **Branch-Isolation auf einem projektpersistenten Workspace** statt Worktrees je Agent, weil Iteration über Läufe hinweg parallele Isolation schlägt (19.3). Worktrees sind seit Release 0.21.0 als Isolationseinheit nicht-schreibender Child Runs zurückgekehrt und seit 0.22.0 auch für schreibende, dort abgesichert durch Workspace Leases — siehe **ADR-6: Workspace Leases und exklusive Ressourcen** sowie 19.10.

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.

Szenario	Strategie
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

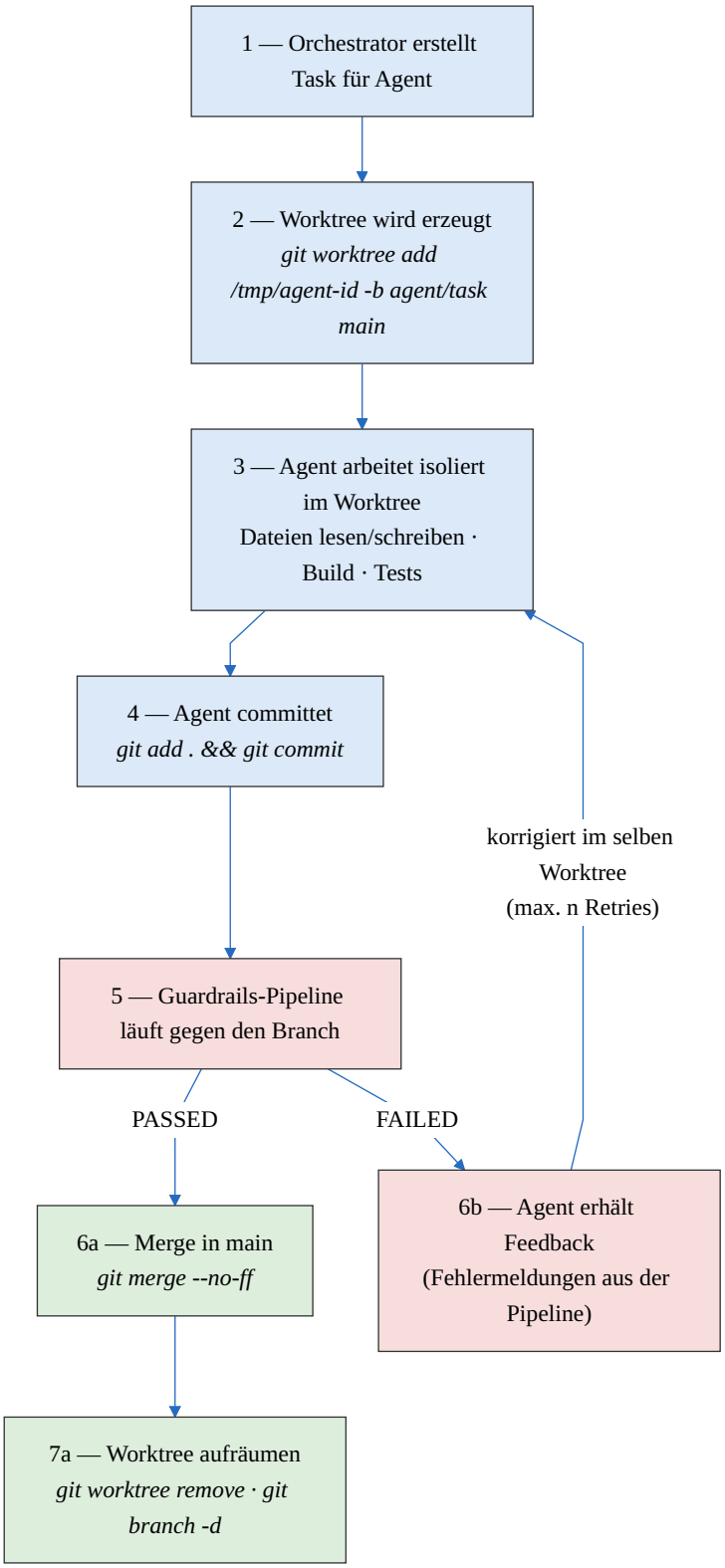

Abbildung 20.2.: Worktree-Lebenszyklus: von der Task-Erstellung bis zu Merge oder Korrektur

- 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.

  Praxis-Check SoftwareFabrik (abweichend): Statt Worktrees je Agent: Branch-Isolation auf einem projektpersistenten Workspace, ergänzt um Prozess- bzw. Container-Isolation. Grund: Iteration über Läufe hinweg schlägt parallele Isolation — der zweite Lauf soll auf dem Ergebnis des ersten aufsetzen. Der Preis (Läufe desselben Projekts nicht beliebig parallel) ist benannt und akzeptiert (19.3, 19.8). *Status (v2.3): Worktrees sind zurückgekehrt — seit 0.21.0 als Isolationseinheit je nicht-schreibendem Child Run, seit 0.22.0 auch für schreibende, dort abgesichert durch Workspace Leases (ADR-6, 19.10).*

ADR-3: Guardrail Pipeline als Pflicht-Gate

Entscheidungsstatus: *Accepted, amended* — Entscheidung vom März 2026 (v1.3), in der Praxis bestätigt und um den Betriebsmodus `advisory` ergänzt (19.5); in der Zielarchitektur wird das Gate zweistufig (lokales Task-Gate je Child Run plus Integration Gate auf dem zusammengeführten Stand) — siehe **ADR-7: Zweistufiges Quality Gate** und 19.10.

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, hard-codierte 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

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

Aspekt	Details
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

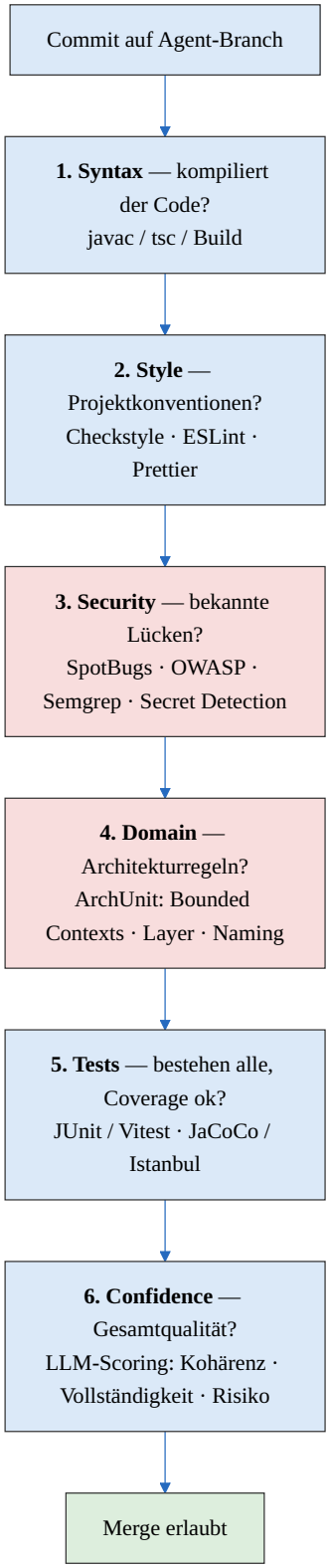

Abbildung 20.3.: Die sechs Pipeline-Stufen der Guardrails-Validierung

Stufe 5: Test-Validierung

Aspekt	Details
Prüfung	Alle bestehenden Tests bestehen (Regression), neue Tests für neuen Code vorhanden, Coverage $\geq$ Schwellwert
Tools	JUnit 5 + Vitest (Execution), JaCoCo + Istanbul (Coverage)
Schwellwerte	Line Coverage $\geq$ 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 nicht abgedeckten Zeilen
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 $< 50 \rightarrow$ 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. Als Planungsannahme — nicht als Messwert — kann von einer Mehrheit der Agenten-Ausgaben ausgegangen werden, die die Pipeline ohne Korrekturbedarf besteht; die tatsächliche First-Pass-Rate ist eine der Messgrößen des Messplans (15.6). Menschliche Reviewer können sich auf die verbleibenden Fälle konzentrieren und auf die Confidence-Score-Begründungen als Entscheidungshilfe zugreifen.

Konsequenzen

Positive Konsequenzen

- Systematischer, reproduzierbarer Prüfprozess für jedes generierte Code-Artefakt – unabhängig davon, welcher Agent oder welches Modell den Code erzeugt hat; die Prüfergebnisse selbst sind wegen der LLM-Stufe teilweise nichtdeterministisch

- 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: Mehrere unterschiedlich arbeitende Prüfungen reduzieren das Risiko, dass ein einzelner Fehler unbemerkt bleibt – eine vollständige Fehlererkennung wird nicht behauptet
- Die ersten fünf Stufen prüfen überwiegend deterministische Kriterien; die LLM-basierte letzte Stufe kann zusätzliche kontextuelle Auffälligkeiten identifizieren, die vom konfigurierten Regelwerk nicht erfasst werden

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).

Praxis-Check SoftwareFabrik (erweitert): Umgesetzt — und um den `advisory`-Betriebsmodus ergänzt: Ein Gate, das am ersten Tag blockiert, wird am zweiten Tag abgeschaltet; der Einführungspfad lautet erst messen, dann durchsetzen. Regulierte Compliance-Profile erzwingen `blocking` (19.5). *Status (v2.1): Die Zielarchitektur macht das Gate zweistufig — lokales Task-Gate je Child Run plus Integration Gate auf dem zusammengeführten Stand (19.10).*

ADR-4: Git-basierte Orchestrierung

Entscheidungsstatus: *Amended* (präzisiert) — Entscheidung vom März 2026 (v1.3), präzisiert im August 2026 (v2.1): Git ist Artefakt-, Versions- und Checkpoint-System; der autoritative **Prozesszustand** liegt in der Datenbank, weil ein Auditor Fragen stellt, die `git log` nicht beantwortet — welche Policy galt, welches Modell arbeitete, wer hat freigegeben (19.8). Siehe **ADR-8: Git als Artefaktzustand, Datenbank als Prozesszustand**.

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.

(Präzisierung v2.1: Git ist Artefakt-, Versions- und Checkpoint-System. Der autoritative Prozesszustand — Status, Policy-Version, Modellauflösung, Freigaben, Gate-Ergebnisse, Audit-Kette — liegt in der Datenbank; siehe ADR-8.)

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

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
```

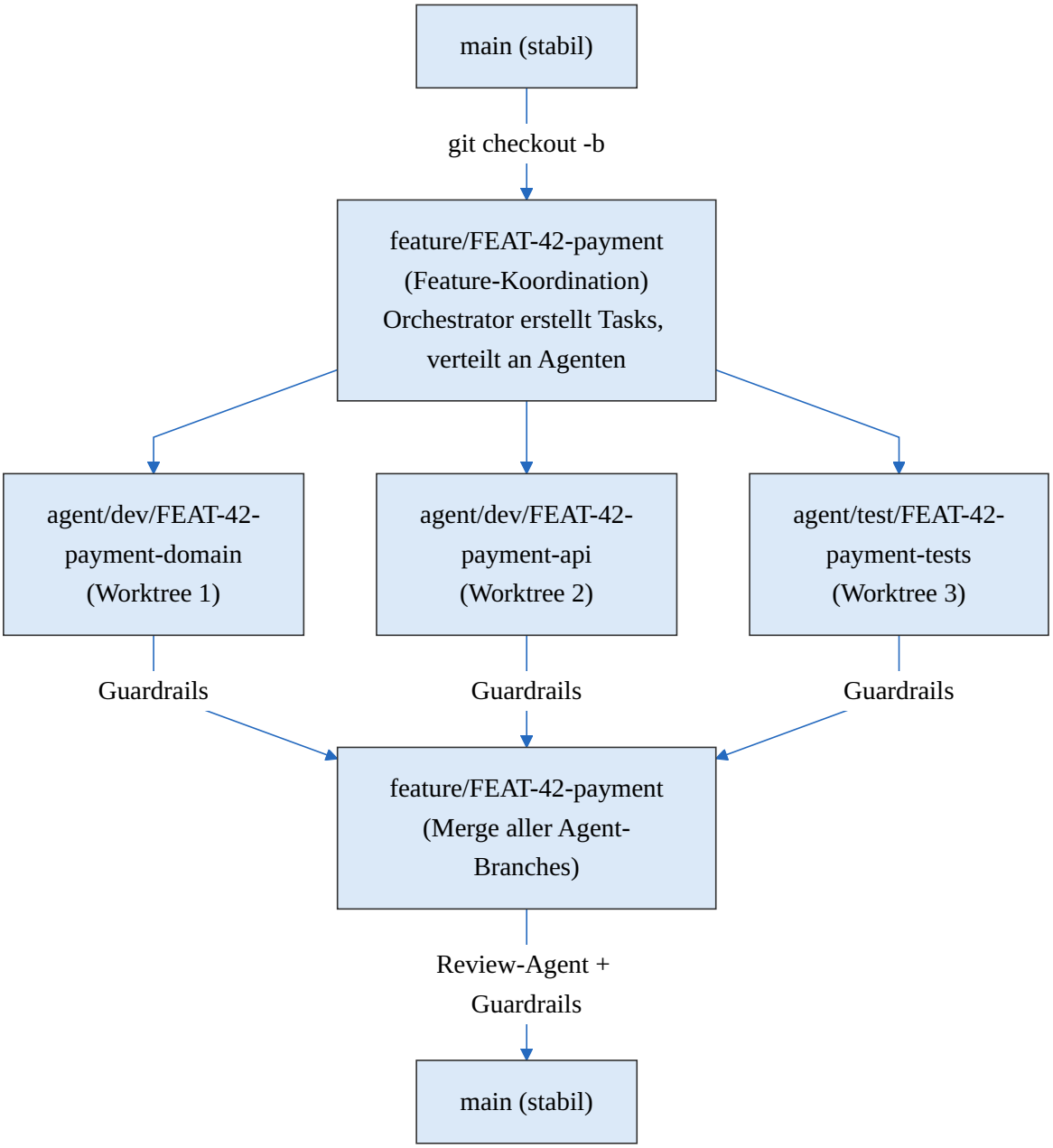

Abbildung 20.4.: Git als State-Machine

```
|   |   +- ...
|   +- 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

```json
{
    "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": "2026-03-04T10:00:00Z",
    "started_at": "2026-03-04T10:00:05Z",
    "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.

Praxis-Check SoftwareFabrik (erweitert): Git ist Zustandsträger — Branch, Commit, Checkpoint, Merge, Pull Request —, aber nicht die alleinige Wahrheit: Der autoritative Zustand liegt in der Datenbank, weil ein Auditor Fragen stellt, die `git log` nicht beantwortet — welche Policy galt, welches Modell arbeitete, wer hat freigegeben (19.8). *Status (v2.6): bestätigt und fortgeschrieben — Workflow-, Task- und Planungszustand liegen seit 0.21.0 autoritativ in der Datenbank, der Lease-Zustand seit 0.22.0 (ADR-8, 19.10).*

ADR-5: Hierarchische Orchestrierung mit Parent Workflow und Child Runs

Entscheidungsstatus: *Accepted* (August 2026) — ersetzt ADR-1. **Implementierungsstatus:** *Implemented* für die Roadmap-Stufen 0 bis 4 und 6 — Parent Workflow, Child Runs, Task-Graph, Synthese, parallel **schreibende** Child Runs mit Merge Queue und Integration Gate, Contract Registry, versionierte Planrevisionen, Merge Intelligence sowie die Produktivitäts- und Qualitätsmessung samt Testabdeckungsänderung, Rollback-Erkennung und Eingriffs- und Aufwandserfassung —, hinter deaktiviertem Feature-Flag, mit zwei Ausnahmen, die ohne Flag wirken (Einzel-Run-Teil der Rollback-Erkennung, Aufwandsfeld an der Run-Freigabe); die Zuordnung der Releases zeigt die Release-Verlaufstabelle in 19.10. Von Stufe 5 ist die Koordinationsschicht umgesetzt — Worker-Registrierung mit Herzschlag, Anspruch vor Seitenwirkung, automatischer Versand hinter eigenem, standardmäßig deaktiviertem Schalter; der verteilte Betrieb über mehrere Hosts ist zurückgestellt, weil die Voraussetzung gemessenen Bedarfs nicht erfüllt ist. Der Zuschnitt der Rollback-Erkennung bestätigt die strikte Richtung dieses ADR: getrennte Anwender für Run- und Workflow-Ebene, weil der `run`-Slice nichts von der Workflow-Ebene wissen darf (19.10).

Motivation / Kontext

ADR-1 entschied sich für sieben spezialisierte, gleichzeitig arbeitende Agenten unter einem Orchestrator. Die Referenzimplementierung hat diese Entscheidung korrigiert: Mehrere gleichzeitig **schreibende** Agenten auf einem Repository erzeugen Konfliktkosten, Zurechnungsprobleme („wer hat das geschrieben?“) und Zwischenstände, die sich nicht attestieren lassen. Der Nutzen mehrerer Agenten liegt in Perspektivenvielfalt — und die ist beim Prüfen wertvoller als beim Erzeugen (19.8).

Gleichzeitig bleibt der ursprüngliche Wunsch berechtigt: Ein Vorhaben, das aus unabhängigen Teilen besteht, sollte nicht künstlich sequenziell abgearbeitet werden.

Entscheidung

Parallelität wird von der Agentenrolle auf die **Workflow-Ebene** verlagert. Ein Parent Workflow zerlegt ein Vorhaben in einen Task-Graph; voneinander unabhängige Tasks werden als isolierte

Child Runs ausgeführt — je Child Run ein eigener Branch oder Worktree und genau ein schreibender Agent (AP-2). Ein Merge Coordinator führt die Ergebnisse kontrolliert zusammen, ein Integration Gate validiert den Gesamtstand.

Konsequenzen

Positiv: echte Parallelität ohne Schreibkonkurrenz; jede Änderung bleibt einem Child Run zurechenbar und attestierbar; die Dekomposition wird explizit statt implizit. Negativ: höhere Control-Plane-Komplexität (Task-Graph, Leases, Merge-Koordination); Integrationskosten können den Parallelitätsgewinn aufzehren, wenn Tasks schlecht geschnitten sind.

ADR-6: Workspace Leases und exklusive Ressourcen

> **Entscheidungsstatus:** *Accepted* (August 2026) — konkretisiert ADR-2 für parallele Child Runs. **Implementierungsstatus:** *Implemented* seit Release 0.22.0: Pfadbereiche in drei Arten (`OWNED`, `READ_ONLY`, `PROTECTED`), Konfliktprüfung auf Segmentgrenzen **vor** dem Start, Leases mit Ablauffrist und Herzschlag, Build-Dateien und Migrationsverzeichnis immer exklusiv (19.10).

Entscheidung

Jeder Child Run erhält eine zeitlich begrenzte **Workspace Lease** über die Pfade und Module, die er beschreiben darf (Owned Paths), ergänzt um read-only und geschützte Bereiche. Ressourcen, die naturgemäß nur einer gleichzeitig ändern kann — Build-Konfiguration, Datenbankmigrationen, gemeinsame API-Spezifikationen, globale Security-Konfiguration —, werden über exklusive Locks serialisiert. Überschneidende Schreibbereiche blockieren die parallele Ausführung **vor** dem Start, nicht erst beim Merge.

Konsequenzen

Positiv: AP-2 wird technisch erzwungen statt organisatorisch erhofft; Konflikte werden zu einer Planungsfrage statt zu einem Merge-Problem. Negativ: Leases brauchen TTL, Heartbeats und definierte Recovery-Zustände — bei unklarem Besitz gilt Fail Closed (AP-7).

ADR-7: Zweistufiges Quality Gate

> **Entscheidungsstatus:** *Accepted* (August 2026) — erweitert ADR-3. **Implementierungsstatus:** *Implemented* seit Release 0.22.0: Das lokale Gate je Child Run prüft eine Änderung für sich, das Integration Gate den zusammengeführten Gesamtstand; nur über dessen Urteil erreicht ein Workflow den Zustand `COMPLETED`. Seit 0.25.0 ordnet eine Konfliktklassifikation die Merge-Konflikte in sieben Arten ein und entscheidet daran, ob ein Rebase überhaupt aussichtsreich ist (19.10).

Entscheidung

Das Pflicht-Gate wird zweistufig: Ein **lokales Task-Gate** prüft jeden Child Run für sich (Syntax, Style, Security, Domain, Tests, Claim Verification, lokaler LLM-Review). Ein **Integration Gate** prüft den zusammengeführten Stand (Gesamtbuild, Regression, End-to-End, Vertragskompatibilität, Migrationskonsistenz, Gesamt-SBOM, Integrationsreview). Erst das Integration Gate schließt einen Workflow ab.

Konsequenzen

Positiv: Fehler werden früh und lokal gefunden, ohne dass der Gesamtnachweis entfällt; ein grünes lokales Gate wird nie mit einem grünen Gesamtstand verwechselt. Negativ: doppelte Prüfkosten und längere Gesamtlaufzeit — der Preis für einen belastbaren Integrationsnachweis.

ADR-8: Git als Artefaktzustand, Datenbank als Prozesszustand

> **Entscheidungsstatus:** *Accepted* (August 2026) — präzisiert ADR-4. **Implementierungsstatus:** *Implemented* für die Run-Ebene (19.6) und seit 0.21.0 auch für Workflow- und Task-Zustände samt Planfreigabe und Workflow-Budget; seit 0.22.0 zusätzlich für Workspace Leases und die Merge Queue, seit 0.26.0 für Worker-Registrierung und Task-Ansprüche (19.10). Die Kennzahlen aus 0.27.0 bestätigen die Entscheidung von der anderen Seite: Sie halten bewusst *keinen* eigenen Zustand, sondern werden vollständig aus dem vorhandenen Prozesszustand abgeleitet — auch die mit 0.28.0 nachgereichten Abdeckungswerte liegen am Build-Ergebnis (V52), die Rollback-Anker aus 0.29.0 an Merge-Queue-Eintrag und Run (V53) und das erfragte Aufwandsfeld aus 0.30.0 an der Freigabeentscheidung (V54), nicht in einer eigenen Kennzahlentabelle.

Entscheidung

Git bleibt Artefakt-, Versions- und Checkpoint-System: Branches, Commits, Diffs, Checkpoints, Pull Requests. Der autoritative **Prozesszustand** liegt dagegen in der Datenbank: Workflow- und Task-Status, Planversionen, Policy-Version, Modellauflösung, Freigaben, Gate-Ergebnisse, Leases und die signierte Audit-Kette.

Konsequenzen

Positiv: Auditierbare Fragen („welche Regel galt zu diesem Zeitpunkt?“, „wer hat freigegeben?“, „welches Modell hat gearbeitet?“) sind beantwortbar — `git log` allein kann das nicht. Negativ: zwei Zustandsträger, die konsistent gehalten werden müssen; die Zuordnung Commit $\leftrightarrow$ Prozessereignis muss explizit persistiert werden.

21. Die Vendor-Frage ist eine Konfigurationsfrage

Hinweis: Dieses Kapitel ersetzt den Werkzeugvergleich der v1.3 („Claude Code vs. andere KI-Entwicklungssysteme“). Die damalige Gegenüberstellung — Claude Code als orchestrierte Multi-Agent-Pipeline, Devin als autonomer Single-Agent, Cursor/Copilot als IDE-Assistenten — war zum Erscheinungszeitpunkt eine brauchbare Landkarte. Sie ist aus zwei Gründen nicht mehr die richtige Frage: Der Markt hat sich vervielfacht und angeglichen, und die relevanten Fähigkeiten sind zu Standards geworden.

21.1. Was sich seit v1.3 verändert hat

Der Markt ist breit und konvergent geworden. Stand August 2026 existiert für jede Kategorie eine Mehrzahl leistungsfähiger Produkte: Terminal-Agenten (u. a. Claude Code von Anthropic, die Codex-CLI von OpenAI, die Gemini CLI von Google, dazu Open-Source-Agenten wie Aider oder OpenCode), IDE-integrierte Assistenten mit Agentenmodus (u. a. Cursor, GitHub Copilot) und autonome Cloud-Agenten (u. a. Devin, dazu die Cloud-Betriebsformen der Terminal-Agenten selbst).[1] Die Fähigkeitsmerkmale, an denen v1.3 die Systeme unterschied — Subagenten, deklarative Repo-Konfiguration, Hooks, Werkzeuganbindung, Hintergrund-Ausführung — finden sich inzwischen in mehreren Produkten. Nach Beobachtung des Autors wechseln einzelne Spitzenplätze auf Benchmarks im Monatsrhythmus; eine Vergleichstabelle wäre bei Drucklegung veraltet.

Die Differenzierungsmerkmale von damals sind Standards geworden. Die deklarative Konfiguration im Repository hat mit `AGENTS.md` ein werkzeugübergreifendes Format (Kapitel 4) [1]; die Werkzeuganbindung hat mit MCP einen herstellerneutralen Industriestandard unter der Linux Foundation (Kapitel 18) [10]. Wer seine Projektregeln, Skills und Werkzeugintegrationen gegen diese Standards baut, kann den darunterliegenden Agenten wechseln, ohne die Investition zu verlieren.

21.2. Die Konsequenz für die Architektur

Damit kehrt sich die Fragestellung um. Nicht mehr: *Welches Werkzeug ist das beste?* Sondern: *Wie baue ich meinen Entwicklungsprozess so, dass die Werkzeugwahl eine Konfigurationsentscheidung ist?* Drei Regeln folgen daraus:

[1] Die Produktzuordnungen lassen sich sämtlich über die Unterstützerliste des AGENTS.md-Standards nachvollziehen [1], die u. a. OpenAI (Codex), Google (Gemini CLI), Cursor, GitHub Copilot, Cognition (Devin), Aider und OpenCode führt; Stand August 2026.

1. **Kein Vendor im Fachkern.** Kein Bestandteil des eigenen Prozesses — Skripte, CI, Governance, Dokumentation — sollte einen konkreten Agenten hart voraussetzen. Die Anbindung gehört hinter eine Schnittstelle.
2. **Standards vor Produkt-Features.** Regeln in `AGENTS.md` statt in werkzeugspezifischen Dateien pflegen, Integrationen als MCP-Server statt als produktspezifische Plugins bauen. Produktspezifische Features (etwa Hooks) bewusst als Zusatz behandeln, nicht als Fundament.
3. **Mehrgleisigkeit einpreisen.** Nach Erfahrung des Autors arbeiten Teams bereits heute mit mehreren Agenten nebeneinander — je nach Aufgabe, Preismodell und Verfügbarkeit. Ein Prozess, der das zulässt, verhandelt bei jedem Modellsprung aus einer Position der Stärke.

Dass diese Umkehrung trägt, ist keine Theorie: Kapitel 19 beschreibt ein System, in dem zehn Agenten-Anbindungen hinter einem einzigen Port stehen und die Vendor-Neutralität als Build-Bedingung maschinell erzwungen wird. Die Adapterwahl ist dort buchstäblich ein Konfigurationsfeld je Lauf.

> **Praxis-Check SoftwareFabrik (erweitert):** Der v1.3-Werkzeugvergleich ist durch die Implementierung hinfällig geworden — die Fabrik integriert zehn Execution-Adapter (ein deterministischer Mock, sechs CLI-Agenten, drei Cloud-Gateways) hinter einem Port; zwei ArchUnit-Regeln machen es technisch unmöglich, Anwendungs- oder Web-Schicht an einen konkreten Vendor zu koppeln (19.2, 19.4).

21.3. Was von der v1.3-Einordnung bleibt

Zwei Aussagen des alten Kapitels haben sich als richtig erwiesen und bleiben gültig: Erstens, dass **Governance das entscheidende Enterprise-Kriterium** ist — daran hat sich nichts geändert, nur dass Governance heute weniger vom einzelnen Werkzeug zu erwarten ist als von der Prozessschicht darüber (Kapitel 12, 14, 19.5–19.6). Zweitens, dass sich **Autonomiegrade unterscheiden** müssen: gesteuerte Human-in-the-Loop-Arbeit, beaufsichtigte Hintergrund-Läufe und autonome Agenten sind verschiedene Betriebsarten mit verschiedenen Risikoprofilen — heute oft innerhalb desselben Produkts wählbar, was die Betriebsart zur bewussten Prozessentscheidung macht statt zur Werkzeugeigenschaft.

22. 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 die Phasen 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.

> **Aufbau dieses Kapitels (v2.2):** Abschnitt 22.3 zeigt den Ablauf so, wie er heute in der Referenzimplementierung stattfindet — ein **Single-Run-Prozess** mit einem schreibenden Agenten und parallelen Reviewern (Teil A, implementiert). Abschnitt 22.4 zeigt denselben Auftrag als **parallelen Workflow** (Teil B, umgesetzt hinter standardmäßig deaktiviertem Feature-Flag; offen ist allein der verteilte Betrieb über mehrere Hosts). Die ursprüngliche v1.3-Skizze mit sieben gleichzeitig arbeitenden Lebenszyklus-Agenten ist damit durch zwei präzise Varianten ersetzt: die, die als Einzel-Run läuft, und die, die als paralleler Workflow läuft.

22.1. 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 durch eine zentrale Control Plane orchestriert. Der Orchestrator strukturiert das Vorhaben in Spezifikation, Planung, Umsetzung, Validierung, Integration und Übergabe. Ob diese Phasen durch einen einzelnen zustandsbehafteten Run oder durch mehrere koordinierte Child Runs ausgeführt werden, hängt vom Architekturstand und von den Abhängigkeiten des Vorhabens ab.

Während der gesamten Ausführung greift die Ausführung 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.

22.2. Gesamtworkflow

Der Gesamtworkflow lässt sich in Spezifikation, Planung, Umsetzung, Validierung, Integration und Übergabe gliedern. Wie diese Phasen technisch ausgeführt werden, hängt vom Architekturstand ab: Die aktuelle Referenzimplementierung verarbeitet den Auftrag als **einen** zustandsbehafteten Run mit einem schreibenden Agenten und mehreren unabhängigen Reviewern. Die Workflow-Ebene zerlegt denselben Auftrag in einen Parent Workflow mit mehreren isolierten Child Runs. Die folgenden beiden Abschnitte stellen beide Varianten gegenüber.

(Die beiden Workflow-Abbildungen der v1.3 an dieser Stelle — „End-to-End Workflow“ und „Interaktion der Systemkomponenten“ — sind in v2.0 mit den inhaltsgleichen Darstellungen in Kapitel 16 und Kapitel 20 konsolidiert.)

22.3. Teil A — der implementierte Single-Run-Prozess

> **Status: implementiert** (Referenzimplementierung, Stand 0.30.0; vgl. Kapitel 19.1–19.3).

So läuft der Auftrag heute. Ein Lauf, ein schreibender Agent, mehrere unabhängige Prüfer:

1. **Spezifizieren.** Aus dem Wizard entstehen versionierte Markdown-Artefakte (Projektbeschreibung, Instruktionen, Guardrails, Definition of Done). Der Mensch editiert sie, bevor der Lauf startet — hier wird Absicht maschinenlesbar.
2. **Anlegen und prüfen.** Der Orchestrator prüft Modell-Policy, erlaubte Adapter, Attestierungspflicht und Budgetgrenzen; die geltende Policy-Version wird attestiert. Verlangt sie eine Freigabe vor der Ausführung, geht der Lauf in `WAITING_FOR_APPROVAL`.
3. **Workspace vorbereiten.** Der projektpersistente Workspace wird auf den Remote-Stand vorgespult, die Spezifikations-Artefakte, das Projektgedächtnis und die Guardrails-Projektion (`AGENTS.md` plus minimale `CLAUDE.md`) werden geschrieben, dann zweigt der Lauf auf einen eigenen Branch ab (`sdlc/run-<id>`).
4. **Ausführen.** Genau **ein** Agentenprozess läuft in der Sandbox auf dem Workspace — Domain, Application und Infrastruktur des Payment Service entstehen nacheinander im selben Lauf. Logs, Token- und Kostenzähler laufen live mit.
5. **Validieren.** Build-Gate (`mvn verify`), Commit auf dem Run-Branch — auch bei Misserfolg, damit die Arbeit inspizierbar bleibt —, optional SBOM, dann das Quality Gate: sechs

Read-only-Reviewer parallel auf dem Diff (zwei LLM-Reviews, Security, Architektur, Claim Verification, Dependency-Scan), aggregiert zu PASS/WARN/FAIL bzw. ERROR.

6. **Korrigieren.** Bei Build-Fehler, blockierendem Gate, Merge-Konflikt oder roter CI wird der Befund als Feedback-Text in einen erneuten Agentenlauf eingespeist — maximal zwei Versuche, danach bleibt der Lauf in `NEEDS_CORRECTION`.
7. **Übergeben.** Lokaler Merge oder Push mit Pull Request; im PR-Fall wartet der Lauf auf grüne CI und Merge. Erst dann gilt das Backlog-Element als erledigt.
8. **Belegen.** Jeder Schritt liegt als signiertes Glied der Audit-Kette vor; der Warum-Trace beantwortet für diesen Payment Service, welche Policy galt, welches Modell gearbeitet hat und wer freigegeben hat.

Die Parallelität liegt hier ausschließlich in Schritt 5 — bei der Bewertung, nicht bei der Erzeugung (AP-2, AP-4).

22.4. Teil B — derselbe Auftrag als paralleler Workflow

> **Status: überwiegend implementiert** — es fehlt allein der verteilte Betrieb über mehrere Hosts, Stufe 5b (vgl. 19.10 und ADR-5 bis ADR-7). Die Workflow-Ebene — Task-Graph, parallele Child Runs in getrennten Worktrees, Planfreigabe, Synthese, Pfad-Besitzmodell, Merge Queue, Integration Gate, Contract Versions, Konfliktklassifikation und Koordinationsschicht — ist vollständig implementiert und steht hinter einem standardmäßig deaktivierten Feature-Flag (der automatische Versand freigewordener Nachfolger-Tasks zusätzlich hinter einem eigenen Schalter, Standard aus); welches Release welchen Baustein brachte, zeigt die Release-Verlaufstabelle in 19.10. Der hier skizzierte Ablauf ist damit vollständig durch implementierte Mechanismen gedeckt. Die Skizze bleibt Pseudocode: Sie zeigt die Struktur, nicht die Aufrufsyntax des Systems.

Der Payment Service besteht aus Teilen, die sich sauber schneiden lassen — genau der Fall, für den die Workflow-Ebene gedacht ist:

```
Parent Workflow: "Payment Service"
   |
   +- Task 1a  Externe Verträge (blockierend)
   |           OpenAPI-Spezifikation · Domain Events · DTO-Schemas
   |
   +- Task 1b  Interne Verträge (blockierend)
   |           Use-Case-Ports · Domain-IDs · gemeinsame Value-Object-
   |           Schnittstellen, als eigenes Contract-Modul
   |           → beide als Contract Version v1 (Content-Hash), attestiert
   |
   +- Task 2   Domain-Implementierung        abhängig von 1b
   |           Owned Paths: domain/**        Child Run A, Branch A
   |
   +- Task 3   API-/Application-Schicht      abhängig von 1a + 1b,
```

```
|           Owned Paths: adapter/in/**,   nicht von der fertigen
|                        application/**
|                                         Implementierung aus Task 2
|                                         Child Run B, Branch B
|
+- Task 4   Infrastruktur (Kafka/Outbox)  abhängig von 1a + 1b
|           Owned Paths: adapter/out/**   (Domain Events aus 1a)
|                                         Child Run C, Branch C
|           Exklusiv-Lock: db/migration/**
|
+- Task 5   Tests                         abhängig von 2-4
|
+- Task 6   Deployment-Artefakte          abhängig von 5
```

Der Ablauf: Die beiden Contract Tasks laufen zuerst und allein — ohne versionierte Verträge keine parallele Implementierung (AP-3). Die Trennung in **externe** und **interne** Verträge ist dabei entscheidend und in der v1.3-Fassung dieses Beispiels noch untergegangen: Eine OpenAPI-Datei und DTO-Schemas allein genügen nicht, damit die Application-Schicht ohne die fertige Domänenimplementierung übersetzt. Erst wenn auch die internen Verträge — Use-Case-Ports, Domain-IDs, gemeinsame Value-Object- Schnittstellen — als eigenes, stabiles Contract-Modul versioniert vorliegen, sind Task 2 und Task 3 wirklich unabhängig. Danach starten die Tasks 2 bis 4 als **Child Runs** mit je eigenem Branch und je genau einem schreibenden Agenten (AP-2); ihre Schreibbereiche überschneiden sich nicht, und die Datenbankmigrationen sind über ein Exklusiv-Lock serialisiert. Jeder Child Run durchläuft sein **lokales Task-Gate**. Der **Merge Coordinator** führt die erfolgreichen Ergebnisse in definierter Reihenfolge auf den Integrationsbranch, rebased auf den jeweils aktuellen Stand und klassifiziert Konflikte; ein Vertragsbruch oder ein veralteter Basisstand setzt den betroffenen Task auf Neuplanung statt auf blindes Retry (AP-1, 19.10). Das **Integration Gate** prüft den zusammengeführten Stand als Ganzes — Gesamtbuild, Regression, End-to-End, API- und Event-Kompatibilität, Migrationskonsistenz, Gesamt-SBOM. Erst danach folgen Freigabe und Pull Request.

Das angestrebte Ergebnis ist eine kürzere Wanduhrzeit bei unveränderter Zurechenbarkeit: Jede Änderung bleibt einem Child Run zugeordnet, jeder Schritt bleibt attestiert, und der Warum-Trace umfasst zusätzlich Planversion, Vertragsversionen und Merge-Entscheidungen. Ob der Parallelitätsgewinn die zusätzlichen Integrationskosten übersteigt, muss der Messplan zeigen (15.6) — Parallelität lohnt sich nur, wenn die Dekomposition trägt (19.10, Risiken).

22.5. 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.

22.6. 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 AGENTS.md beziehungsweise 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

Wie die Pipeline auf Befunde reagiert, hängt vom Betriebsmodus des Gates ab. Im Blocking-Modus beziehungsweise unter einem strikten Kontrollprofil kann der Workflow erst fortgesetzt werden, wenn alle vorgeschriebenen Prüfungen bestanden wurden. Im Advisory-Modus werden die Befunde protokolliert und angezeigt, ohne den Übergang automatisch zu blockieren; im Modus `off` findet keine Prüfung statt (19.5). Nur der Blocking-Modus erzwingt für agentisch erzeugten Code definierte Mindestmaßstäbe an Qualität, Sicherheit und Architektur — und auch dann als Risikoreduktion, nicht als Garantie.

22.7. Deployment-Ergebnis

Das End-to-End-Szenario zeigt, wie Spezifikation, Agentenausführung, unabhängige Prüfung, Governance und Übergabe in einem konsistenten Prozess verbunden werden können. Ein agentisches Entwicklungssystem ist damit mehr als ein Werkzeug zur Codegenerierung: Es ist die Prozessschicht, die diese Schritte zusammenhält.

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

- klare Trennung zwischen erzeugender und prüfender Verantwortung
- nachvollziehbare Entwicklungsabläufe mit reproduzierbarem Prüfprozess
- frühzeitige Validierung von Architektur-, Sicherheits- und Qualitätsanforderungen
- bessere Wiederverwendbarkeit von Wissen durch Shared Knowledge Store und ADRs
- potenziell kürzere Durchlaufzeit bei kontrollierter Qualitätssicherung; der tatsächliche Vorteil gegenüber einem Einzel-Run ist Gegenstand des Messplans (15.6)

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.

> **Praxis-Check SoftwareFabrik (bestätigt):** Der hier skizzierte End-to-End-Pfad ist dort reproduzierbar produktisiert: Wizard (18 Templates inklusive Repo-Import) → versionierte Spezifikations-Artefakte → Run → Quality Gate → Merge bzw. Pull Request (19.1).

23. Troubleshooting & Schnellreferenz

Versionshinweis (v2.0): Die Schnellreferenz nennt v1.3-Werkzeugbegriffe. Heute gilt: `maxTurns` in der Subagenten-Definition statt `max_turns` im Aufruf, Fortsetzung über `SendMessage` statt `resume`, Agent-Tool statt Task-Tool (vgl. 3.5).

Problem	Lösung
Unvollständige Ergebnisse	Prompts aufteilen. `maxTurns` in der Subagenten-Definition erhöhen.
Falsche Dateien geändert	Pfade explizit angeben. `isolation="worktree"`.
Kontextverlust	Nachricht per SendMessage an die Agent-ID senden.
Parallele Kollisionen	Worktree-Isolation für alle parallelen Agenten.
Halluzinierte APIs	Guardrail-Hooks aktivieren. Für kritische Tasks eine stärkere Modellklasse bzw. höhere Fähigkeitsstufe wählen (vgl. Kap. 5).
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	Development Capability (Modellwahl über Capability-Routing)
Tests schreiben/ausführen	test-agent
Code-Qualität prüfen	Review Capability (Modellwahl über Capability-Routing)
Deployment	deploy-agent

Aufgabe	Agent
CI/CD Quality Gate	qa-guard

Praxis-Check SoftwareFabrik (erweitert): Aus der Schnellreferenz wurden betriebliche Abläufe (Demo-Betrieb, Air-Gap-Auslieferung) und dokumentierte Praxis-Fallstricke — etwa der Architektur-Ratchet mit Zyklen-Kappungsgrenze oder CI-Jobs, die sich ohne Secret still überspringen und grün wirken (19.2, 19.5, 19.7).

24. Glossar & Abkürzungsverzeichnis

Abkürzung / Begriff	Erklärung
ADR	Architecture Decision Record – Dokumentierte Architekturentscheidung.
AOP	Aspect-Oriented Programming – Querschnittsbelange (Logging, Security).
AP-1 bis AP-8	Die acht architektonischen Prinzipien der Control-Plane-Architektur (Kapitel 2).
Attestierung	Signierter, verketteter Nachweis einer Entscheidung im Agentensystem (Kap. 19).
Blatt-Slice	Modul, das nur konsumiert und von keinem anderen konsumiert wird — Entwurfsmuster zur Zyklenvermeidung (Kap. 19).
BPMN	Business Process Model and Notation – Geschäftsprozessmodellierung.
Build-Run	Lauf, der ein Backlog-Element umsetzt: Code auf isoliertem Branch, validiert, gemergt oder als Pull Request (Kap. 19).
Child Run	Ein regulärer Run, der einem Workflow Task zugeordnet ist (seit 0.21.0 für nicht-schreibende, seit 0.22.0 auch für schreibende Tasks implementiert, Feature-Flag — Kap. 19.10).
CI/CD	Continuous Integration / Deployment – Automatisierte Pipeline.
Claim Verification	Prüfung der Aussagen eines Agenten über seine eigene Arbeit, etwa „alle Tests bestehen“ (in v1.3: Halluzinationserkennung; Kap. 12, 19.5).
Contract Version	Versionierter bzw. gehashter gemeinsamer Vertrag, gegen den mehrere Child Runs arbeiten (seit 0.23.0 implementiert, Feature-Flag; Kap. 19.10).

Abkürzung / Begriff	Erklärung
Control Plane	Steuerschicht, die Agentenläufe orchestriert, begrenzt und protokolliert, ohne selbst Code zu schreiben (Kap. 19).
DAG	Directed Acyclic Graph – Gerichteter azyklischer Graph (Task Graph).
DDD	Domain-Driven Design – Fachliche Softwaremodellierung.
Debt-Ratchet	Eingefrorene, gezählte Architektur-Altschuld; neue Verstöße brechen den Build, behobene verschwinden aus der Baseline (Kap. 19).
DTO	Data Transfer Object – Datenobjekt ohne Geschäftslogik.
Guardrails-Projektion	Materialisierung der versionierten Verhaltensregeln als AGENTS.md/CLAUDE.md je Lauf in den Workspace (Kap. 19).
HPA	Horizontal Pod Autoscaler – Kubernetes-Skalierung.
IaC	Infrastructure as Code – Deklarative Infrastruktur.
Integration Gate	Quality Gate auf dem zusammengeführten Workflow-Stand (seit 0.22.0 implementiert, Feature-Flag; Kap. 19.10).
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).
Mandant	Isolationseinheit für Projekte, Läufe, Budgets und Policies in einer mehrmandantenfähigen Plattform (Kap. 19).
MCP	Model Context Protocol – herstellerneutraler Industriestandard zur Werkzeuganbindung (ursprünglich Anthropic, seit 12/2025 Agentic AI Foundation / Linux Foundation; Kap. 18).
Merge Coordinator	Komponente zur kontrollierten Rebase-, Merge-, Konflikt- und Revalidierungssteuerung (seit 0.22.0 implementiert, Feature-Flag; Kap. 19.10).

Abkürzung / Begriff	Erklärung
OWASP	Open Worldwide Application Security Project (bis 2023: Open Web Application Security Project) – Sicherheitsstandards.
Owned Paths	Die Pfade, die ein Workflow Task exklusiv beschreiben darf — technische Grundlage des Single-Writer-Prinzips (Kap. 22.4, 19.10).
Plan-Run	Lauf, der nur analysiert und Arbeitsschritte vorschlägt; darf keinen Code ändern (Kap. 19).
Policy-as-Code	Regeln als versioniertes, signiertes Dokument mit genau einer aktiven Version je Mandant (Kap. 19).
PSD2	Payment Services Directive 2 – EU-Zahlungsdiensterichtlinie.
Quality Gate	Aggregation der Reviewer-Befunde zu einer Entscheidung (PASS/WARN/FAIL, bei Prüferausfall ERROR — nie ein stiller Pass) (Kap. 12, 19).
RBAC	Role-Based Access Control – Rollenbasierte Zugriffskontrolle.
Replanner	Komponente, die den Task Graph auf Basis neuer Informationen versioniert anpasst (seit 0.24.0 implementiert, Feature-Flag; Kap. 19.10).
RFC 9457 (vormals RFC 7807)	Problem Details for HTTP APIs – Strukturierte Fehlerantworten.
RTM	Requirements Traceability Matrix – Anforderungsverfolgung.
Run	Zustandsbehaftete Ausführungseinheit eines Agentenauftrags mit Phasen, Zuständen und Korrekturschleife (Kap. 19).
SBOM	Software Bill of Materials – Software-Komponentenliste.
SDLC	Software Development Lifecycle – Entwicklungslebenszyklus.
Sealed Interface	Java-Feature zur Einschränkung implementierender Klassen.
Single Writer	Prinzip: Innerhalb eines Branch, Worktree oder Workspace arbeitet genau ein Agent schreibend (Kap. 19).

Abkürzung / Begriff	Erklärung
Slice	Fachlich geschnittenes Modul (Bounded Context) eines modularen Monolithen (Kap. 19).
Testcontainers	Docker-basierte Integrationstests für Java.
Warum-Trace	Rekonstruktion je Lauf, welche Policy, welches Modell und welche Freigaben gewirkt haben (Kap. 19).
Workflow	Übergeordnete, zustandsbehaftete Ausführungseinheit für ein Feature oder Vorhaben (seit 0.21.0 implementiert, Feature-Flag; Kap. 19.10).
Workflow Task	Abgegrenzte Arbeitseinheit mit Abhängigkeiten, Capability, Schreibbereichen und optionalem Child Run (seit 0.21.0 implementiert, die Schreibbereiche seit 0.22.0; Feature-Flag — Kap. 19.10).
Workspace Lease	Zeitlich begrenzte Reservierung von Pfaden, Modulen oder exklusiven Ressourcen (seit 0.22.0 implementiert, Feature-Flag; Kap. 19.10).

Literatur

[1] *AGENTS.md — a simple, open format for guiding coding agents.* Agentic AI Foundation / Linux Foundation. URL: https://agents.md/ (besucht am 6. August 2026).

[2] *Authentication — Claude Code Documentation.* Anthropic. URL: https://code.claude.com/docs/en/authentication (besucht am 6. August 2026).

[3] *Codex CLI — Documentation.* OpenAI. URL: https://developers.openai.com/codex/cli/ (besucht am 6. August 2026).

[4] *Create custom subagents — Claude Code Documentation.* Anthropic. URL: https://code.claude.com/docs/en/sub-agents (besucht am 6. August 2026).

[5] *deepset-mxbai-embed-de-large-v1 — German/English Embedding Model.* Mixedbread AI / deepset. URL: https://huggingface.co/mixedbread-ai/deepset-mxbai-embed-de-large-v1 (besucht am 6. August 2026).

[6] *Introducing the Model Context Protocol.* Anthropic. 25. November 2024. URL: https://www.anthropic.com/news/model-context-protocol (besucht am 6. August 2026).

[7] *Linux Foundation Announces the Formation of the Agentic AI Foundation.* The Linux Foundation. 9. Dezember 2025. URL: https://www.linuxfoundation.org/press/linux-foundation-announces-the-formation-of-the-agentic-ai-foundation (besucht am 6. August 2026).

[8] *MCP joins the Agentic AI Foundation.* Model Context Protocol Project. 9. Dezember 2025. URL: https://blog.modelcontextprotocol.io/posts/2025-12-09-mcp-joins-agentic-ai-foundation/ (besucht am 6. August 2026).

[9] *Migration guide — Claude Developer Platform.* Anthropic. URL: https://platform.claude.com/docs/en/about-claude/models/migration-guide (besucht am 10. August 2026).

[10] *Model Context Protocol — Specification and Documentation.* Agentic AI Foundation / Linux Foundation. URL: https://modelcontextprotocol.io/ (besucht am 6. August 2026).

[11] *Models overview — Claude Developer Platform.* Anthropic. URL: https://platform.claude.com/docs/en/about-claude/models/overview (besucht am 6. August 2026).

[12] *Pricing — Claude Developer Platform.* Anthropic. URL: https://platform.claude.com/docs/en/about-claude/pricing (besucht am 6. August 2026).

[13] *Verordnung (EU) 2022/2554 des Europäischen Parlaments und des Rates vom 14. Dezember 2022 über die digitale operationale Resilienz im Finanzsektor (DORA)*. 14. Dezember 2022. URL: `https://eur-lex.europa.eu/eli/reg/2022/2554/oj` (besucht am 6. August 2026).

[14] *Verordnung (EU) 2024/1689 des Europäischen Parlaments und des Rates vom 13. Juni 2024 zur Festlegung harmonisierter Vorschriften für künstliche Intelligenz (KI-Verordnung)*. 13. Juni 2024. URL: `https://eur-lex.europa.eu/eli/reg/2024/1689/oj` (besucht am 6. August 2026).