Die SoftwareFabrik

Von der Referenzarchitektur zum System

Eine Control Plane für den kontrollierten Einsatz von Coding-Agenten — Aufbau, Governance und Grenzen einer Referenzimplementierung

Sonderdruck

Kapitel 19 des Whitepapers
Agentic Software Development — Enterprise Architecture with AI Agents

Martin Janda

martin@janda.io

Sonderdruck zur Whitepaper-Version 3.2 · SoftwareFabrik 0.39.0 · **Stand: 20. Oktober 2026**

Lizenz: CC BY 4.0

Inhaltsverzeichnis

Zu diesem Sonderdruck **4**

19 Die SoftwareFabrik — von der Referenzarchitektur zum System **5**

19.1 Von der Referenzarchitektur zur Implementierung 6
Der Kernablauf 6
Kennzahlen der Codebasis 6
Bewusste Nicht-Ziele 7
19.2 Systemkontext und Bausteinsicht 8
Modularer Monolith mit erzwungenen Grenzen 8
Das Domänenmodell: der Entwicklungsprozess als Fachdomäne 11
19.3 Das Ausführungsmodell: der Run 12
Drei Run-Arten, sieben Phasen, 14 Zustände 12
Ablauf eines Build-Runs 13
Der Regelkreis: Korrekturschleife statt Retry 15
19.4 Vendor-Neutralität als Architektureigenschaft 15
Abo-Modus statt Token-Abrechnung 17
Capability-Routing statt Modellnamen 17
Sandbox — eine Korrektur 18
Werkzeuge nur auf Freigabe: MCP-Nullzustand, Isolationsnachweis, Gateway . . 18
19.5 Guardrails in der Praxis 20
Die Gate-Policy 20
Drei Betriebsmodi 21
19.6 Governance und Nachweisführung 21
Mandanten und Rollen 22
Policy-as-Code 22
Compliance-Profile 23
Die signierte Audit-Hashkette 24
Warum-Trace und Audit-Export 25
Lieferkette und Wissensbestand 26
Gedächtnis ohne Vektorspeicher 27
19.7 Betrieb und Souveränität 27
19.8 Was die Praxis am Konzept korrigiert hat 29
(1) Ein Agent je Lauf, mehrere Prüfer — statt Hub-and-Spoke-Multi-Agent . . . 29
(2) Regelkreis statt Retry — Repository-Realität als Eingabe 29
(3) Governance ist keine Ergänzung, sondern Struktur 30
(4) Kubernetes ist keine Voraussetzung — Souveränität schon 30

19.9 Grenzen und offene Punkte . . . 30
Threats to Validity . . . 33
19.10 Vom Run zum parallelen Workflow: Ausbaustufen und Umsetzungsstand *(Roadmap)* . . . 34
Der Release-Verlauf 0.21.0 bis 0.39.0 . . . 34
Das Zielbild . . . 35
Die tragenden Prinzipien . . . 36
Die Ausbaustufen . . . 36
Die neue Position . . . 45
19.11 Spec-Driven Development: die Anforderungsebene *(seit 0.35.0)* . . . 45
Die Lücke: gut ist nicht dasselbe wie richtig . . . 45
Die Produkt-Spec . . . 46
Freigabe als eigenes Gate . . . 46
Traceability über IDs, nicht über Texte . . . 47
Conformance: prüfen gegen die Spec . . . 47
Spec-Drift: was eine Änderung veralten lässt . . . 48
Spec-Run: Anforderungen erarbeiten statt nur verwalten . . . 48
Governance: Spec-Pflicht, Nachweis, Kennzahlen *(0.39.0)* . . . 49
Einordnung . . . 50

Literatur **51**

Zu diesem Sonderdruck

Dieser Sonderdruck enthält Kapitel 19 des Whitepapers *Agentic Software Development — Enterprise Architecture with AI Agents* in unverändertem Wortlaut. Das Whitepaper beschreibt eine Referenzarchitektur für den kontrollierten Einsatz von Coding-Agenten; dieses Kapitel beschreibt das System, das daraus entstanden ist. Wer nur wissen will, wie die Architektur in der Praxis aussieht, findet hier alles Nötige — wer wissen will, warum sie so aussieht, findet die Begründung im vollständigen Dokument.

Drei Hinweise zum Lesen:

- Die **Kapitel- und Abschnittsnummerierung** folgt dem Whitepaper. Deshalb beginnt dieser Sonderdruck mit Kapitel 19 und nicht mit Kapitel 1: Der Text verweist durchgängig auf seine eigenen Abschnitte (19.1 bis 19.11), und diese Verweise sollen gültig bleiben.
- **Verweise auf andere Kapitel** sind als solche gekennzeichnet („Kapitel 12 des Whitepapers“). Sie führen in das vollständige Dokument und sind für das Verständnis dieses Kapitels nicht erforderlich.
- **Verweise auf ADR-1 bis ADR-8** meinen die Architekturentscheidungen in Kapitel 20 des Whitepapers. Sie halten fest, welche Entscheidung der Referenzarchitektur die Praxis bestätigt, präzisiert oder ersetzt hat.

Das vollständige Whitepaper steht auf `janda.io` zur Verfügung.

19 Die SoftwareFabrik — von der Referenzarchitektur zum System

Hinweis: Die Konzeptkapitel dieses Whitepapers — Kapitel 1–18 sowie die Architekturentscheidungen (Kapitel 20 des Whitepapers) und der Werkzeugvergleich (Kapitel 21 des Whitepapers) — 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 20. Oktober 2026 auf dem Release-Stand 0.39.0), 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, Stand 0.39.0; als opt-in gekennzeichnete Funktionen sind standardmäßig ausgeschaltet) 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). Abschnitt 19.11 beschreibt die seit 0.35.0 hinzugekommene Anforderungsebene (Spec-Driven Development); sie ist implementiert, aber ebenfalls opt-in hinter einem eigenen, standardmäßig deaktivierten Feature-Flag. 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. Seit 0.35.0 können zusätzlich versionierte, freigegebene Produkt-Specs mit nummerierten Akzeptanzkriterien die Anforderungen tragen (opt-in, 19.11).
3. **Ausführen.** Ein *Run* durchläuft sieben Phasen. Der Coding-Agent läuft als Host-Prozess mit frischem Home-Verzeichnis und ohne nicht freigegebene Werkzeugserver (19.4) 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)	502
Produktivcode	~54.900 Zeilen
Testklassen	360
Fachliche Slices (Module)	34
Datenbanktabellen	63
Flyway-Migrationen	61
Execution-Adapter	10
Review-Adapter	7
Wizard-Templates	18
Coverage-Gate	Line $\geq$ 85 %, Branch $\geq$ 81 % (JaCoCo, buildbrechend)
Releases	47 (0.1.0 bis 0.39.0, April–Oktober 2026)

Erhoben auf dem Release-Tag 0.39.0 (20. Oktober 2026), Anwendungsmodul; die eigenständigen Module Offline-Verifier (~1.100 Zeilen) und MCP-Gateway (~2.400 Zeilen) sind nicht mitgezählt. Der Zuwachs der Releases 0.21.0 bis 0.30.0 geht fast vollständig auf die Workflow-Ebene zurück, der von 0.31.0 bis 0.39.0 auf die Absicherung der Agentenläufe gegen fremde Werkzeugkonfiguration, den unabhängig prüfbaren Nachweis und Spec-Driven Development — welches Release was brachte (einschließlich der Migrationen V40–V65; die Nummern V55 und V57 sind nicht vergeben, 0.39.0 kam ohne Migration aus), dokumentiert die Release-Verlaufstabelle in 19.10. Die vier seit 0.30.0 hinzugekommenen Fachlichkeiten sind `canary`, `mcp` *und* `mcpgateway` *(19.4) sowie* `spec` *(19.11).*

Der Technologiestack ist bewusst konservativ: Java 25, Spring Boot 4.1, 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.

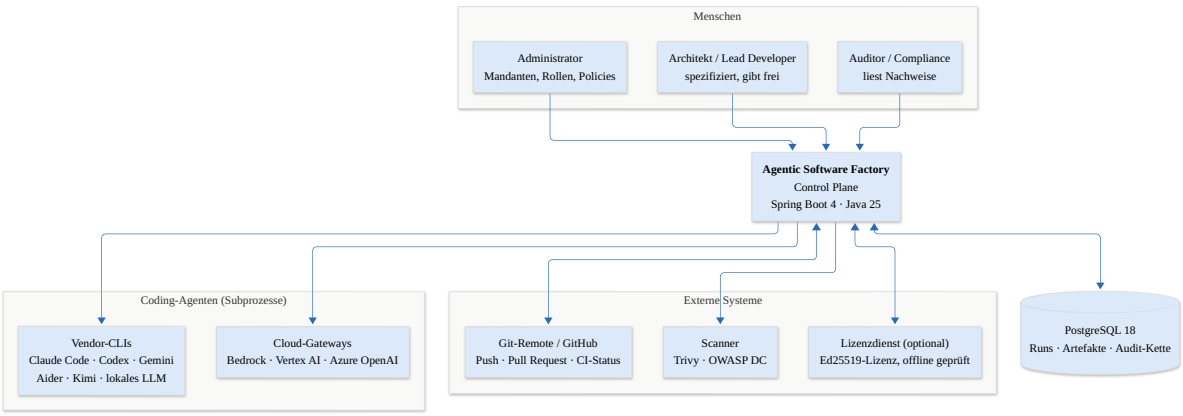


Abbildung 19.1: Systemkontext der Agentic Software Factory: eine Control Plane zwischen Mensch, Coding-Agenten und Zielrepository

- **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. Gegenüber dem Systemkontext aus Kapitel 3 des Whitepapers 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 34. 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]

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

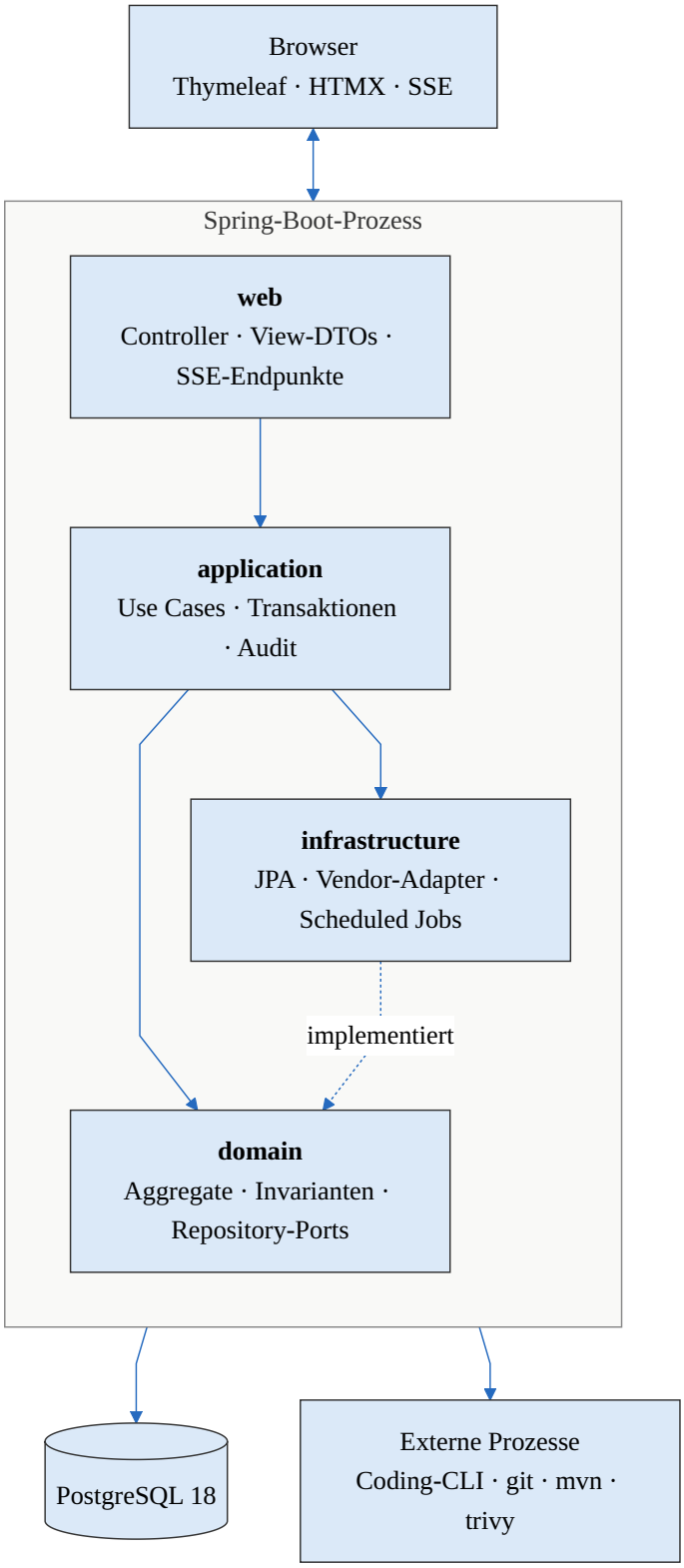


Abbildung 19.2: Bausteinsicht: modularer Monolith mit Ports-and-Adapters pro Slice; externe Werkzeuge ausschließlich hinter Ports

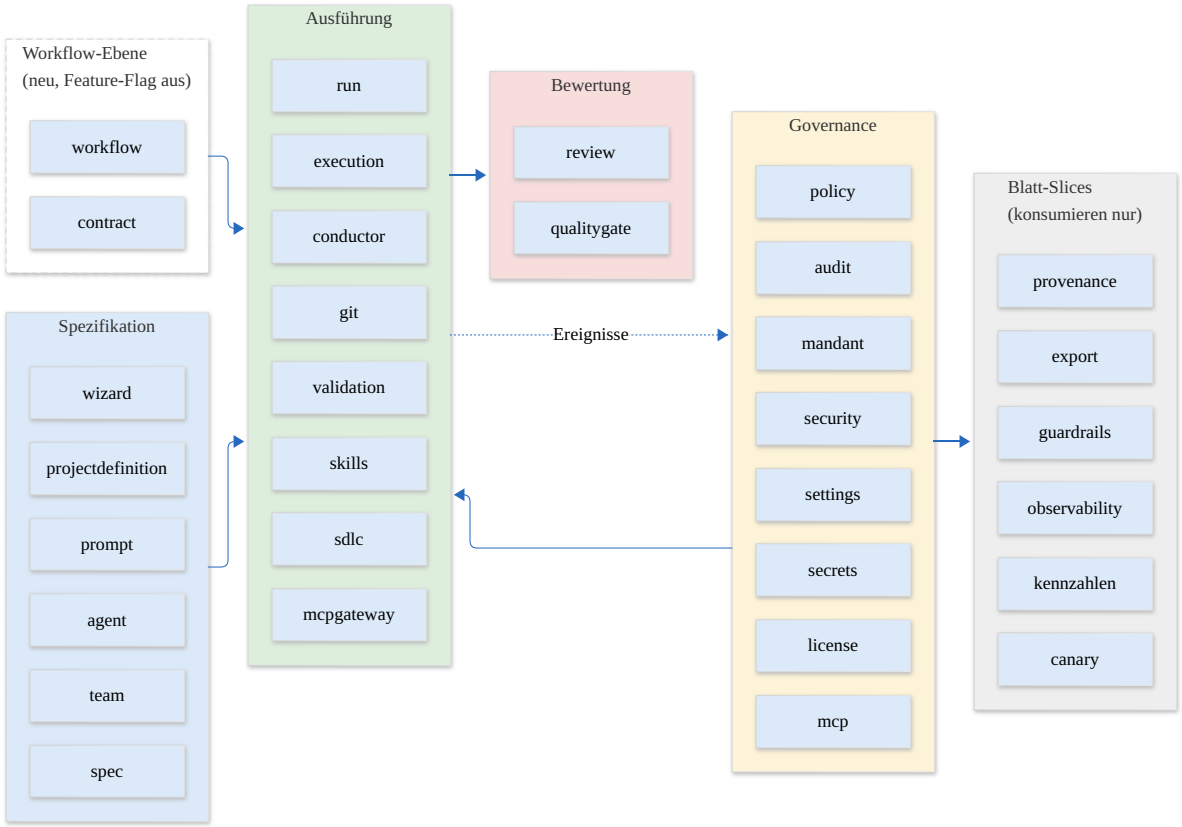


Abbildung 19.3: Fachliche Slices der Fabrik, gruppiert nach Aufgabe. Blatt-Slices (provenance, export, guardrails, observability, kennzahlen, canary) werden von keinem Kern-Slice konsumiert; die technischen Querschnittspakete common und web sind nicht dargestellt

- **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.

Der Ratchet hat einen lehrreichen Fallstrick: Der Freeze speichert nur eine begrenzte Zahl elementarer Modulzyklen (Kappungsgrenze, in ArchUnit standardmäßig 100; die Fabrik hat sie auf 400 angehoben — wirksam erst seit 0.29.0, weil der Name der Einstellung vorher falsch geschrieben war und nichts bewirkte; die eingefrorene Baseline enthält genau 100 Zyklen). 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

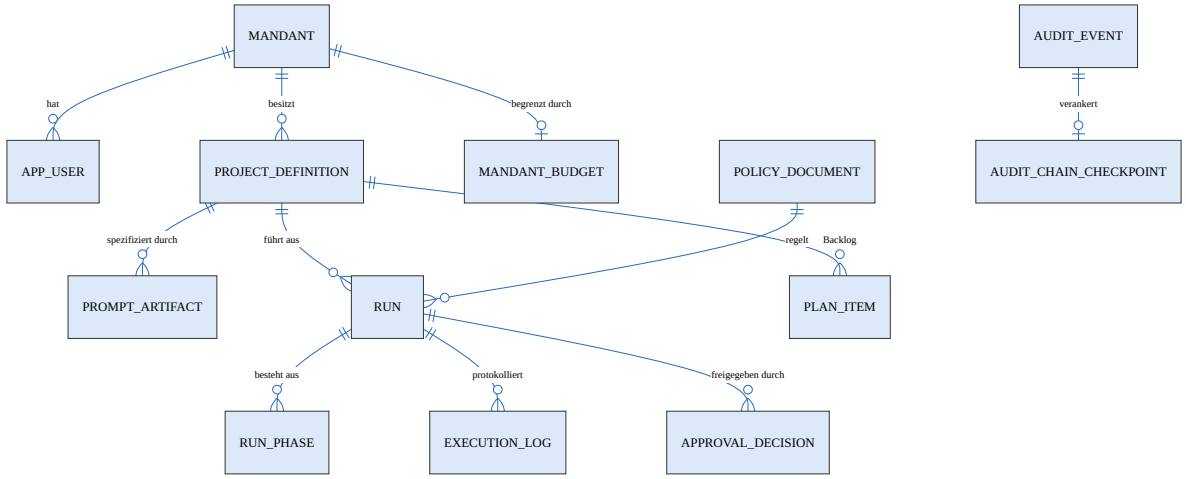


Abbildung 19.4: Kernaggregate der Run-Ebene; vollständig umfasst das Schema 63 Tabellen in 61 Flyway-Migrationen

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 ein Slice außer dem Audit-Export 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 des Whitepapers 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 (Werkzeug), 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, Routi-

nen, 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, V56 das signierte Register der Isolationsnachweise, V58–V60 die Kennzeichnung geschätzter Kosten, das Aufwandsfeld an der Planfreigabe und zusätzliche Kostenfelder, V61–V65 die Anforderungsebene (Specs, Fassungen, Freigaben, Spec-Bezug von Backlog und Workflow-Tasks, Veraltet-Markierung, Rückfragen). 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 des Whitepapers, 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]

Drei Run-Arten, sieben Phasen, 14 Zustände

Ursprünglich gab es 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. Seit 0.38.0 gibt es als dritte Art den **Spec-Run**, von dem nach demselben Muster nur Ergebnisse unter `specs/` übernommen werden und der Anforderungen statt Code erarbeitet (19.11).

Jeder Run durchläuft sieben Phasen (Intake, Prompt-Assembly, Workspace-Preparation, Execution, Validation, Correction, Completion) und bewegt sich durch 14 Zustände; der vierzehnte, `WAITING_FOR_INPUT`, kam mit dem Spec-Run hinzu und ist ihm vorbehalten (19.11). Drei Zustände 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.

[2] `RunOrchestrationService` (~2.600 Zeilen), `RunStatusTransitions`; unerlaubte Übergänge werfen `InvalidRunStateTransitionException`.

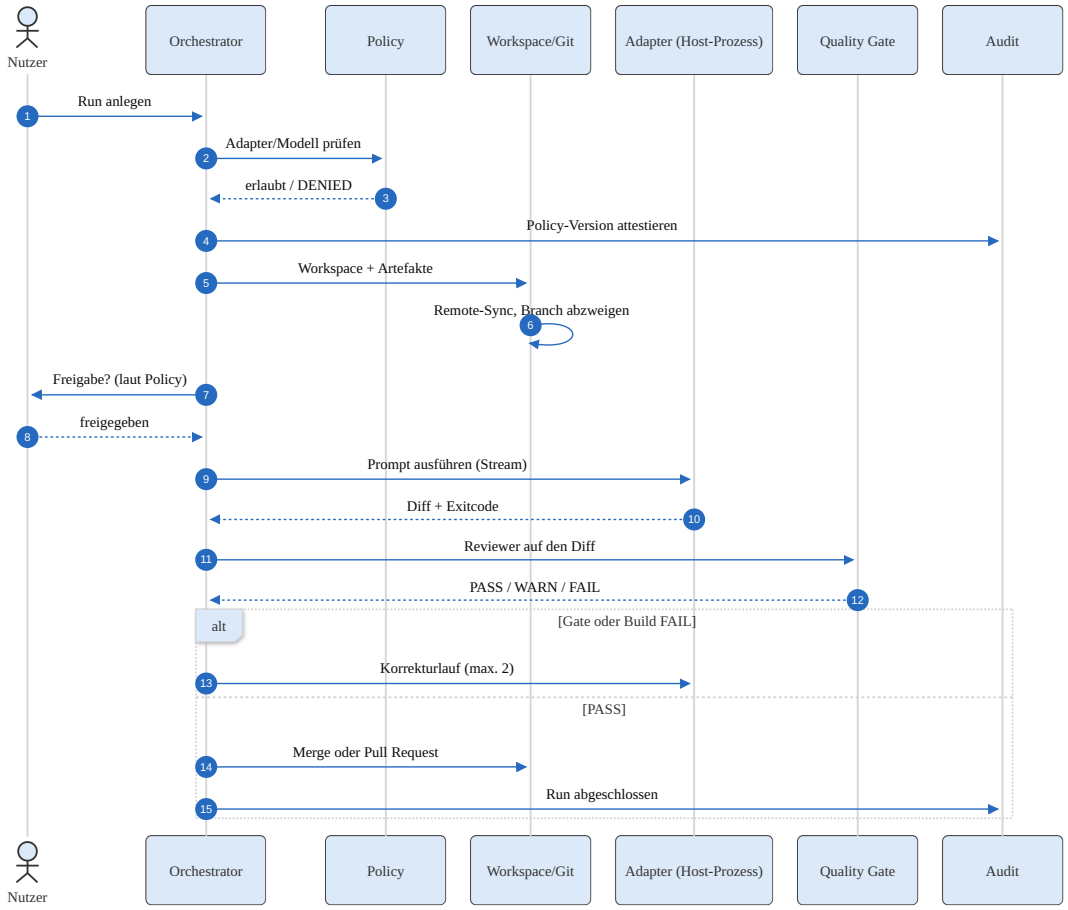


Abbildung 19.5: Laufzeitablauf eines Build-Runs von der Anlage bis zum Merge, inklusive Korrekturschleife und Approval-Punkten

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 Policy-Prüfungen erzeugen ein attestiertes Ereignis `RUN_POLICY_DENIED`, auch die Ablehnung ist Nachweis (Lizenzgrenzen lehnen ohne Audit-Eintrag ab). *Korrektur:* Für die Prüfungen vor der Anlage gilt das erst seit 0.39.0. Bis dahin schrieb die Fabrik das Ereignis in derselben Datenbanktransaktion wie die Ablehnung — und rollte es mit ihr zurück; die Audit-Kette zeigte diese Ablehnungen (Modell-Policy, Policy-as-Code, MCP-Standardverbot, Attestierungspflicht) nie. Ablehnungen *während* des Laufs — unmittelbar vor der Ausführung, bei der Gateway-Anbindung, für gesperrte Prüfer im Gate — standen schon vorher in der Kette, weil der Lauf dort regulär als gescheitert endet. Frühere Fassungen dieses Whitepapers beschrieben die Anlageprüfung im Soll-, nicht im Ist-Zustand. Seit 0.39.0 wird der Eintrag nach Abschluss der umgebenden Transaktion in einer eigenen geschrieben, bewusst nicht verschachtelt, damit beide nicht gegenseitig auf die lückenlose Sequenznummer der Kette warten; scheitert dieses Nachschreiben, wird das proto-

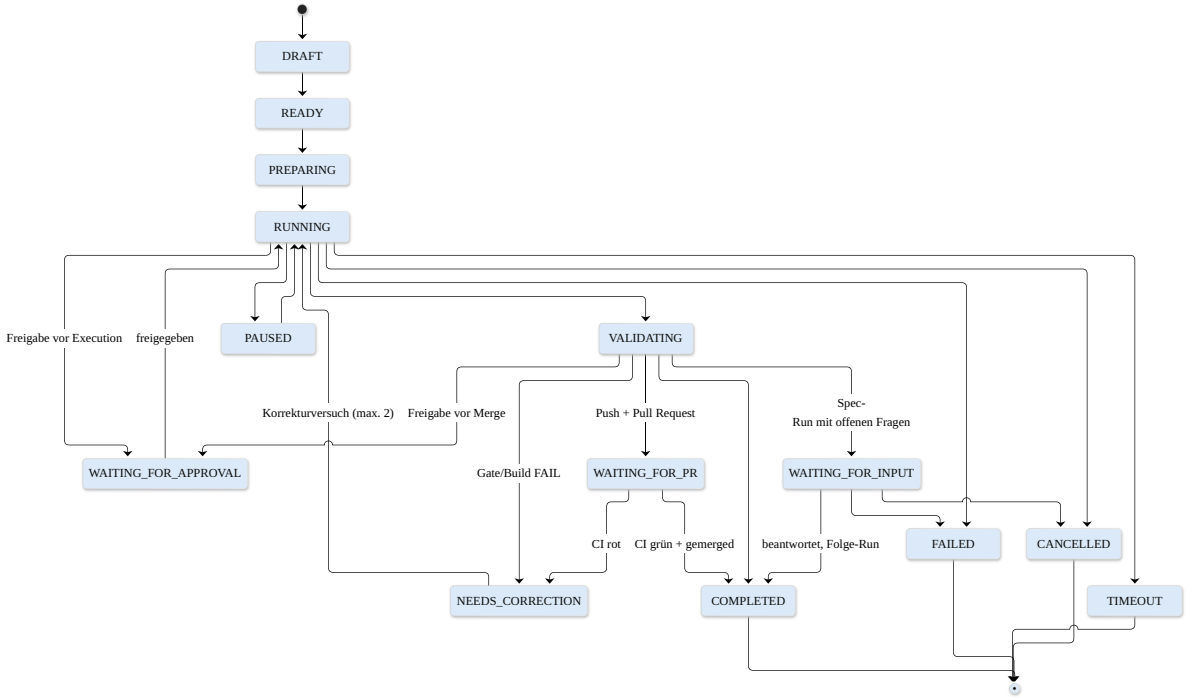


Abbildung 19.6: Zustandsmodell eines Runs. Alle Übergänge sind zentral hinterlegt; unerlaubte Übergänge werfen eine Ausnahme

kolliert, nicht wiederholt.[3] 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* →

[3] `AuditService.erfasseUnabhaengig`; genutzt von `RunOrchestrationService` und `WorkflowService`.

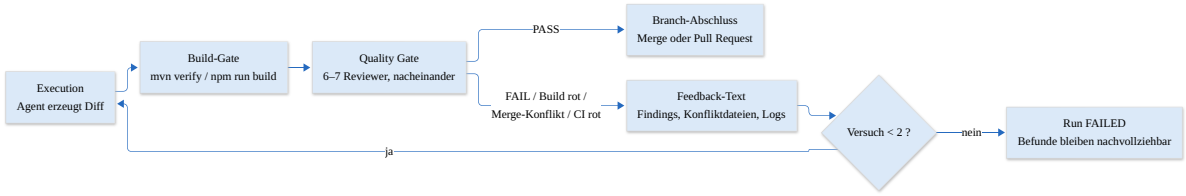


Abbildung 19.7: Die Korrekturschleife als Regelkreis — Befunde werden zu Eingaben des nächsten Laufs

prüfen → *mergen* → *neu planen*, mit definierten Stellen, an denen ein Mensch eingreifen muss.

Der Regelkreis: Korrekturschleife statt Retry

Kapitel 7 des Whitepapers 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 des Whitepapers (in Version 1.3: Kapitel 20 des Whitepapers) vergleicht Werkzeuge gegeneinander. Die Fabrik hat die Frage architektonisch aufgelöst: Die gesamte Vendor-Neutralität hängt an einer Schnittstelle,[4] 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:

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

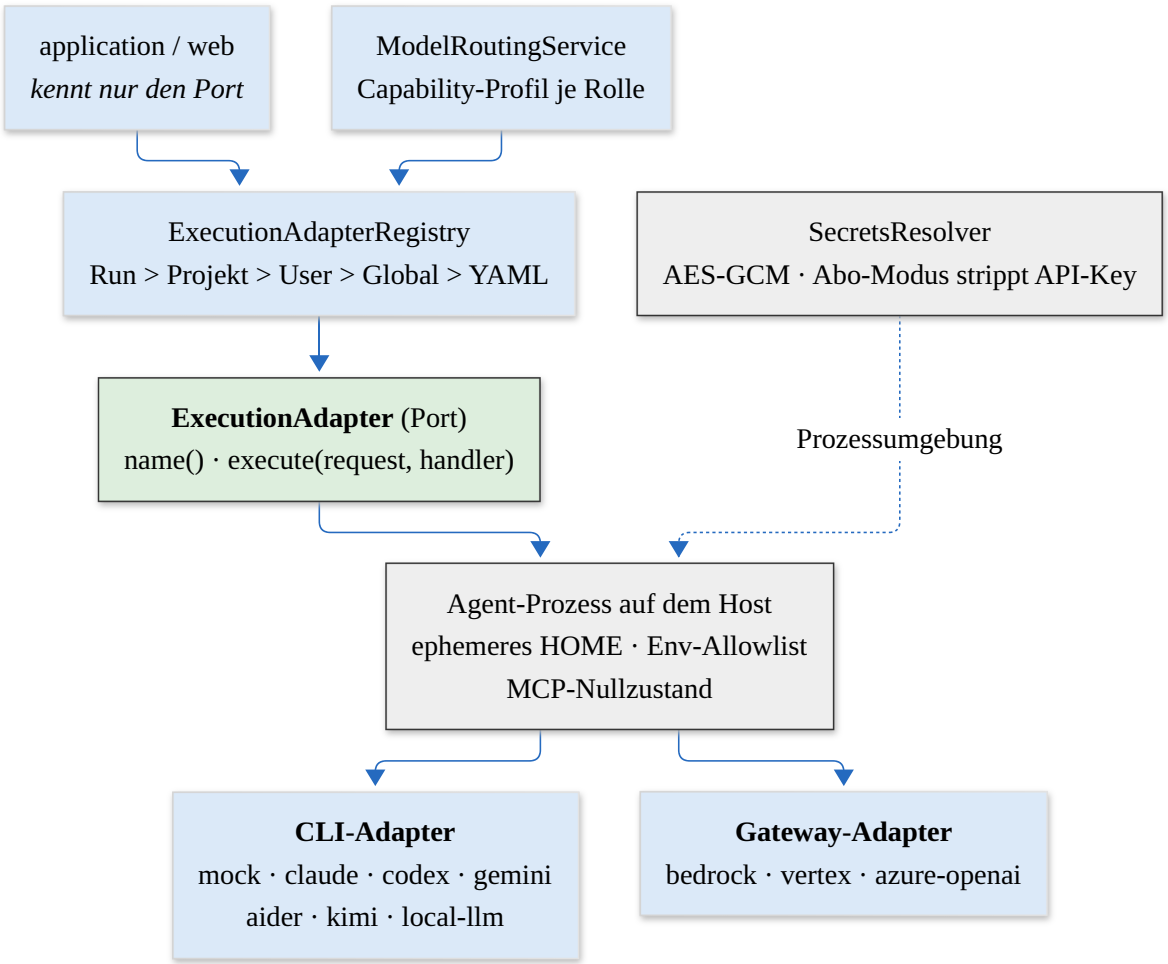


Abbildung 19.8: Vendor-Neutralität durch einen Port: Application- und Web-Schicht kennen ausschließlich den ExecutionAdapter (ArchUnit-erzwungen)

1. **Streaming statt Rückgabewert.** Ein Event-Consumer liefert Logzeilen, Token-Verbrauch und Phasensignale *während* des Laufs — ohne das wäre Live-Beobachtbarkeit unmöglich. Der Port erlaubt das; der Claude-Code-Adapter nutzt es derzeit nicht zeilenweise, sondern sammelt die JSON-Ausgabe der CLI bis zum Prozessende. Und dass ein Abbruch tatsächlich wirkt, war nicht selbstverständlich: Vor 0.33.0 setzte er nur den Status, die CLI lief bis zum Ende oder Timeout weiter; seither beendet eine Prozessregistrierung den Agentenprozess samt Nachkommen.
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 nicht-deterministischen 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 des Whitepapers ist grundlegend: Bei Flatrate-Abos entstehen keine Token-Kosten je Lauf; ein reines Token-ROI-Modell bildet die Wirtschaftlichkeit agentischer Entwicklung nicht mehr vollständig ab. Seit 0.33.0 ist der Abo-Modus ausdrücklich als *persönliche* Nutzung gefasst — der Abo-Inhaber auf seinem eigenen Rechner; der Teambetrieb sperrt den Abo-Login und erzwingt den API-Key-Modus. Eine zuvor angekündigte Variante, bei der mehrere Nutzer ihr jeweils eigenes Abo auf einem gemeinsamen Server einbringen, wurde gestrichen, weil der Anbieter Drittanwendungen das Sammeln, Speichern oder Vermitteln seiner Zugangsdaten untersagt (Stand Oktober 2026).

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. Dass das kein theoretisches Risiko ist, zeigte sich vor 0.34.0: Läufe mit Claude Code wurden durchgehend mit 0 € bewertet, weil der Verbrauchsparser ein Feld auswertete, das die CLI gar nicht liefert — ohne Modell keine Preiszeile, ohne Preiszeile null, und keine Anzeige meldete einen Fehler. Behoben ist der Parser (seither werden auch die gesondert berechneten Cache-Schreib-Tokens erfasst und die Kostenschätzung der CLI als Gegenprobe im Warum-Trace geführt), nicht die Regel: Unbekannte Modelle ergeben weiterhin 0 €. Sauberer wäre ein expliziter Kostenstatus `UNKNOWN`, der die Budgetprüfung nicht bestehen kann, ein konfigurierbarer Sicherheitsersatzwert — und im Abo-Modus ein eigener Status statt einer Zahl. Den letzten Teil hat 0.33.0 in abgeschwächter Form umgesetzt: Abo-Läufe tragen in Warum-Trace, Kostenansichten und Exporten die Kennzeichnung „Schätzung (API-Tarif), nicht abgerechnet“ und zählen bewusst weiter gegen die Budgets, weil sie die erzeugte Last abbilden. Der Rest bleibt ein Punkt für die Weiterentwicklung (19.9).

Capability-Routing statt Modellnamen

Die Modellwahl-Strategie aus Kapitel 5 des Whitepapers 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 die Orchestrierung einen Modellnamen kennt — wichtig in einem Markt, in dem Modellnamen eine Halbwertszeit von Monaten haben. Die Namen stehen an genau einer Stelle, einem Modellkatalog samt Preistabelle und Rollen-Defaults, den eine feste Release-Checkliste gegen die Herstellerübersicht prüft. Eine je Projekt gesetzte Modell-Policy wird beim Anlegen eines Runs erzwungen

und attestiert; im Nachhinein ist belegbar, welches Modell wirklich gearbeitet hat — seit 0.33.0 auch, woher die Wahl kam (Override, Team, Rollen-Default oder Adapter-Default).

Sandbox — eine Korrektur

Die Fabrik kennt zwei Isolationsvarianten, und die Fassungen dieses Whitepapers bis 3.0 haben ihre Reichweite falsch beschrieben. Richtig ist (Stand 0.38.0): Die Coding-Agenten selbst laufen in **keiner** Variante in einem Container, sondern immer als Prozess auf dem Host — mit einer Allowlist der Umgebungsvariablen, die den Agenten überhaupt erreichen, und seit 0.31.0 mit einem frischen Home-Verzeichnis je CLI-Aufruf (siehe unten). Die per Einstellung aktivierbare **Container-Sandbox** kapselt nur die Schritte, die die Fabrik am erzeugten Projekt ausführt: Build und Tests, SBOM-Erzeugung und Abhängigkeits-Scan — dort mit CPU-, Speicher- und Prozesslimits, read-only-Dateisystem und einem Bindmount ausschließlich auf den Workspace, aber **mit** Netzzugang, weil Build und Scanner Abhängigkeiten beziehungsweise die Schwachstellendatenbank laden. Die Containervorlage steht zwar auf `--network=none`, bei keinem dieser Schritte kommt diese Voreinstellung aber zum Tragen. Die netzlose *Agent*-Sandbox, die frühere Fassungen beschrieben, gibt es also nicht. Agent-CLIs im Container — mit projizierten Zugangsdaten und einem Netz, das nur die Modell-Schnittstellen erreicht — sind ein eigenes, offenes Arbeitspaket (19.9).

Aufgedeckt hat das die Fabrik selbst: Bei der Vorbereitung der MCP-Absicherung (unten) fiel auf, dass ein Starthinweis im Container-Modus eine Isolation der Agenten nahelegte, die der Code nie hergestellt hatte; der Hinweis nennt seither in beiden Varianten, was gekapselt ist und was nicht.[5] Die Lehre gleicht der übersprungenen Prüfung in 19.9 (4): Eine Einstellung, die Schutz verspricht, muss belegen, wovor sie schützt — sonst erzeugt sie Vertrauen, das sie nicht deckt.

Zur Absicherung der Außenkontakte gehört eine **Host-Allowlist für Git-Remotes**:[6] 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 des Whitepapers hatte die Bedrohungsklassen richtig benannt, die konkreten Angriffsflächen zeigten sich erst im gebauten System.

Werkzeuge nur auf Freigabe: MCP-Nullzustand, Isolationsnachweis, Gateway

Kapitel 18 des Whitepapers beschreibt MCP-Server und Hooks als Erweiterungspunkte von Agenten. Für eine Steuerschicht sind sie zunächst Angriffsfläche: Coding-CLIs laden Werkzeugserver und Hooks aus mehreren Quellen — aus der Konfiguration des Betriebsnutzers, aus Plugins und aus Dateien im Repository selbst. Bei der Analyse fiel auf, dass Claude Code im nicht-interaktiven Modus eine `.mcp.json` im Workspace ohne Vertrauensrückfrage startete (beobachtet an einer CLI-Version vom September 2026); ein importiertes Repository hätte damit beim ersten Lauf Kommandos auf dem Host ausführen können. Dasselbe galt für Hooks aus den

[5] `SandboxScopeAdvisory` und `ContainerProcessSandbox` im `execution`-Slice; die Korrektur ist in einer Architekturentscheidung der SoftwareFabrik festgehalten.

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

Projekt-Einstellungen eines Repositorys.

Seit 0.31.0 gilt deshalb ein **Nullzustand**: Kein Agentenlauf erreicht einen Werkzeugserver, den die Fabrik nicht ausdrücklich bereitstellt. Jeder CLI-Aufruf — Ausführung wie Review — bekommt ein frisches Home-Verzeichnis, in das nur die Anmeldedateien des jeweiligen Adapters kopiert werden; Claude Code läuft zusätzlich mit strikter, leerer MCP-Konfiguration und seit 0.34.0 ohne Projekt-Einstellungen und mit abgeschalteten Hooks.[7] Vor der ersten Ausführung wird der Konfigurationsbestand des Workspace in die Audit-Kette geschrieben, vorgefundene Server und ausführbare Konfiguration ebenso; schreibt der Agent selbst Werkzeugkonfiguration, wird das attestiert und ist unter dem weiter unten beschriebenen MCP-Standardverbot ein blockierender Security-Befund im Gate, der in die Korrekturschleife läuft. Plugins sind seither standardmäßig gesperrt und werden nur per Policy freigegeben. Die Nebenwirkung ist gewollt: Agentenläufe sehen die Host-Konfiguration nicht mehr, und Commits, die ein Agent selbst anlegt, tragen eine neutrale Identität.

Ob eine CLI diese Einstellungen tatsächlich beachtet, ist eine Annahme, die sich mit jeder CLI-Version ändern kann. Der **Canary-Harness** (0.31.0) prüft sie je Adapter und *installierter* CLI-Version in zwei Läufen gegen eine nachgestellte Host-Umgebung mit Marker-Servern: Ein Kontroll-Lauf mit dem Produktivaufruf ohne Isolation *muss* die Marker zeigen — sonst hätte der Test nichts bewiesen und gilt als ungültig —, der Lauf über den echten Adapter-Pfad darf keinen zeigen.[8] Das Ergebnis wird signiert in einem Register abgelegt und attestiert; anerkannt wird nur der jüngste gültig signierte Eintrag je Adapter und Version. Die Profile BAIT/MaRisk/DORA und BSI/VS-NfD setzen ein Policy-Feld *MCP-Standardverbot*: Dort starten Agent-CLIs nur mit bestandenem Nachweis für die installierte Version, geprüft bei der Anlage und erneut unmittelbar vor der Ausführung; die LLM-Reviewer werden ohne Nachweis als gesperrt übersprungen, die Sperre wird attestiert. Nach dem Update auf 0.31.0 war in diesen Profilen deshalb zunächst nur der Mock-Adapter nutzbar — gewollt, *fail closed*. Was sich nicht markieren lässt, bleibt benannt ungeprüft: systemweite Konfiguration (findet sich dort ein Server-Eintrag, scheitert der Nachweis) und kontogebundene Connectoren des Anbieters. Auch die Wirkung der Hook-Abschaltung ist bisher aus der Herstellerdokumentation abgeleitet, nicht durch den Canary belegt.

Erst auf diesem Nullzustand öffnet der **MCP-Gateway** (0.34.0, opt-in, Standard aus) gezielt: ein eigener Prozess ohne Signaturschlüssel, der Agenten ausschließlich als lesend gekennzeichnete Werkzeuge freigegebener Server bereitstellt — zunächst nur für die Ausführung mit Claude Code, Reviewer und andere Adapter bleiben im Nullzustand. Freigegeben wird über Policy-as-Code; jeder Server ist mit Hashwerten über Startbefehl beziehungsweise Adresse und über seine Werkzeugliste gepinnt, eine Abweichung sperrt den Server und bricht den Lauf ab. Jeder Werkzeugaufruf landet als signiertes Ereignis in der Audit-Kette — mit Hashwerten von Argumenten und Ergebnis statt der Inhalte —, und kann der Gateway ein Ereignis nicht zustellen, führt er für diesen Lauf keinen weiteren Aufruf aus. *Einschränkung:* Ohne Agent-Container kann ein Agent unter demselben Betriebskonto grundsätzlich auf dessen Prozesse zugreifen; die Übergabe von Geheimnissen über die Standardeingabe und Tokens je Lauf verengen diesen Weg, schließen ihn aber erst mit dem Container.

[7] `AgentHome` im `execution`-Slice; Workspace-Prüfung `McpWorkspacePruefung` im `mcp`-Slice.
[8] `CanaryService` im Blatt-Slice `canary`; Register `IsolationsnachweisRegister` im `mcp`-Slice.

19.5 Guardrails in der Praxis

Die Guardrails-Pipeline aus Kapitel 12 des Whitepapers 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.

Sieben Review-Adapter prüfen denselben Diff unabhängig voneinander — technisch nacheinander, nicht gleichzeitig: 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), seit 0.38.0 dazu ein regelbasierter Spec-Conformance-Reviewer, der nur bei Läufen mit Spec-Bezug anspringt und prüft, ob die gebundenen Akzeptanzkriterien in Tests referenziert werden (19.11). 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 des Whitepapers — treffender als **Claim Verification** bezeichnet —, verschoben vom Code auf die Selbstauskunft des Agenten — dort sitzt der klassische Selbstbetrug agentischer Systeme. *Einschränkung:* Die Prüfung auf unberührte Akzeptanzkriterien lief bis einschließlich 0.34.0 ins Leere, weil die Orchestrierung den Reviewern gar keine Kriterien übergab — das Feld war fest leer. Erst die Anforderungsebene aus 19.11 füllt es, und auch nur für Läufe mit Spec-Bezug.

Die Gate-Policy

Das Quality Gate aggregiert die Befunde über eine konfigurierbare Policy[9] 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*,

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

Attestierung ohne Schlüssel ist ein Startfehler — und als Projektregel: ein CI-Job, der sich ohne Secret selbst überspringt, gilt nicht als bestanden).

Wie leicht diese Regel trotzdem unterlaufen wird, zeigte eine Prüfung der Claude-Anbindung vor Release 0.33.0. Sie fand vier Stellen, an denen die Fabrik Erfolg meldete, ohne ihn belegt zu haben. Die LLM-Reviewer werteten einen Fehler-Exit ohne Startfehler (Authentifizierung, Rate-Limit, Verweigerung) oder Freitext ohne Befundzeile als „Review ohne Befunde“, also als bestanden. Das Gate las den Diff des Arbeitsverzeichnisses, obwohl Build-Runs ihre Arbeit vorher auf den Run-Branch committen — es prüfte einen leeren Diff. Ein Abbruch beendete den Agentenprozess nicht (19.4). Und Build-Runs liefen ohne Schreibberechtigung, sodass der Agent im Text um Erlaubnis bat, statt zu arbeiten. Seither gilt ein Review nur als bestanden, wenn der Prozess erfolgreich endet *und* ein nachweisbares Ergebnis liefert (mindestens eine Befundzeile oder die nun verpflichtende Zusammenfassungszeile), sonst wird es zu `ERROR`; das Gate prüft den kumulierten Branch-Diff ab der Abzweigung; Build-Runs erhalten eine eng begrenzte Werkzeugfreigabe ohne Commit, Push und Download-Werkzeuge, und Reviews laufen technisch schreibgeschützt.[10] Die Sonderregel stand also im Code — ihre Voraussetzung, dass ein Ausfall überhaupt als Ausfall erkannt wird, nicht.

Das Confidence Scoring aus Kapitel 12 des Whitepapers 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.

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?**

[10] Changelog 0.33.0 der SoftwareFabrik; Prozessbeendigung über den Port `ProzessRegistry`.

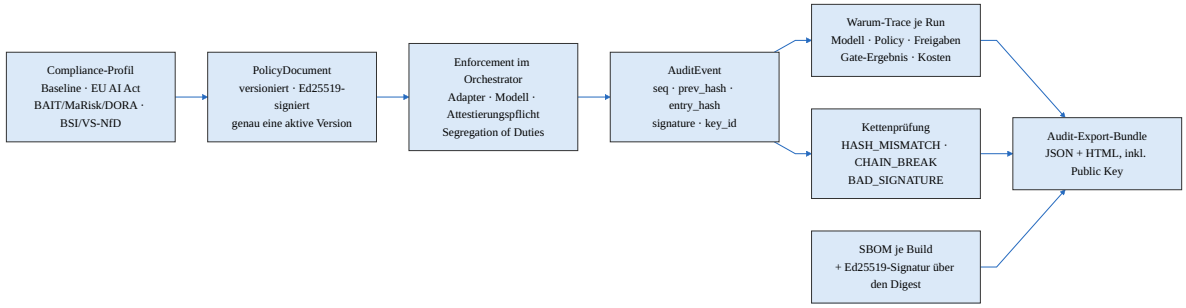


Abbildung 19.9: Von der Policy bis zum Auditbericht: jede Durchsetzung erzeugt ein signiertes Kettenglied

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. Wie wichtig es ist, die Isolation für jede *neue* Ebene erneut nachzuweisen, zeigte Release 0.34.0: Die Workflow-Ebene ließ bis dahin jede angemeldete Rolle jede Aktion ausführen — auch auf Workflows und Verträgen fremder Mandanten, die ID genügte —, und neue Workflows blieben ohne Mandanten. Der festgeschriebene Test deckte die Projekt-Grenze ab, die später hinzugekommene Ebene lag daneben. Betroffen war nur der Betrieb mit eingeschaltetem Workflow-Feature-Flag; seither erben Workflows und Verträge den Mandanten ihres Projekts, und ändernde Aktionen verlangen die passende Rolle.

Das fünfstufige Rollenmodell (Viewer < Developer < Maintainer < Owner < Admin) entspricht dem Least-Privilege-Modell aus Kapitel 14 des Whitepapers; 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; seit 0.31.0 und 0.34.0 dazu das MCP-Standardverbot, die erlaubten Plugins und die erlaubten MCP-Server (19.4), seit 0.39.0 die Spec-Freigabe als Startbedingung und eine Mindestabdeckung der Akzeptanzkriterien (19.11). Neue Felder erscheinen in der kanonischen Form nur, wenn sie vom Standard abweichen — bestehende Policy-Signaturen bleiben dadurch gültig.[11] Je Mandant existiert **genau eine aktive**

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

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) [4]
BAIT / MaRisk / DORA	blocking	Execution, Validation	ja	BaFin-Anforderungen an die IT; DORA (EU) 2022/2554 — IKT-Risiko, Nachweisführung [3]

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. Die Profile BAIT/MaRisk/DORA und BSI/VS-NfD setzen seit 0.31.0 zusätzlich das MCP-Standardverbot: Agent-CLIs laufen dort nur mit bestandenem Isolationsnachweis (19.4). Seit 0.39.0 verlangen die Profile EU AI Act und BAIT/MaRisk/DORA außerdem die Spec-Freigabe: Ein Build startet dort nur gegen eine freigegebene Anforderungsfassung (19.11; wirksam nur, wo Spec-Driven Development freigeschaltet ist). Baseline und BSI/VS-NfD bleiben unverändert — für Verschlusssachen ist die Spec-Pflicht projektspezifisch über die Policy zu setzen. **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.[12] 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

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

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. Innerhalb der Plattform ist das unverändert offen (19.9); der seit 0.32.0 verfügbare Offline-Verifier (unten) trennt für Exporte bereits zwischen „bewiesen“, „Verletzung“ und „nicht vollständig beweisbar“.

Attestiert wird nicht jeder Tastendruck, sondern **jede Entscheidung, die den Handlungsspielraum des Agenten festgelegt hat**: Run-Lebenszyklus, Modellauflösung, angewendete und verweigerte Policies (die Ablehnungen vor der Anlage eines Laufs erst seit 0.39.0, siehe 19.3), angewendete Guardrails-Version, Gate-Ergebnis, Freigaben, erzeugte 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.[13] 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.

Bis einschließlich 0.31.0 hatte dieser Export eine grundsätzliche Schwäche: Er enthielt das Urteil der Fabrik über sich selbst. Je Ereignis standen nur Kurzhash, Aktion und die Markierung „intakt“; zum Nachrechnen fehlten Felder, exportiert wurde die *aktuell* aktive statt der im Lauf angewandten Policy-Version, und wer Einträge fälschen kann, kann auch den mitgelieferten öffentlichen Schlüssel austauschen. Release 0.32.0 macht den Export deshalb **ohne Zugriff auf die Fabrik prüfbar**. Ein eigenständiger **Offline-Verifier** — ein Programm von heute rund 1.100 Zeilen, nur auf dem JDK aufgebaut, ohne Bibliotheken und ohne Code der Fabrik, reproduzierbar gebaut und getrennt von der Installation verteilt, damit ein Prüfer das Werkzeug nicht von der geprüften Partei bezieht — rechnet Hash und Signatur jedes Eintrags nach, prüft die lückenlose Verkettung bis zu einem signierten Ketten-Kopf, die im Lauf angewandten Policy-

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

Versionen und je Run ein signiertes Manifest der zugehörigen Ereignisse.[14] Die Frage, die eine Prüfung zuerst stellt — wurde etwas weggelassen? —, löst der Export über **Hash-Stellvertreter**: Alle Einträge außerhalb der exportierten Runs bis zum Ketten-Kopf — darunter die anderer Mandanten — erscheinen nur als Hashwerte ohne Inhalt; die Verkettung bleibt prüfbar, ohne fremde Daten offenzulegen. Den Vertrauensanker bringt der Prüfer selbst mit: den Fingerabdruck des Signaturschlüssels, den der Betreiber auf einem zweiten Weg veröffentlicht. Ohne ihn endet die Prüfung nie mit „bewiesen“, sondern mit „intern konsistent, Vertrauen nicht verankert“. Die Grenze ist dokumentiert: Wer vor dem Export Datenbankzugriff hat, kann die jüngsten Einträge löschen und den Kopf auf einen älteren, gültig signierten Stand zurücksetzen; dagegen hilft nur der Abgleich mit einem früher bekannten Kettenstand, den der Verifier entgegennimmt.

Release 0.39.0 erweitert den Export um die Anforderungsebene (19.11). Schema 2.1 enthält je gebundener Spec-Fassung Inhalt, Hash und Freigabe (wer, wann); den Hash signiert die Fabrik beim Export mit dem Attestierungsschlüssel — dieselbe Vorschrift wie bei den Policy-Versionen. Der Verifier 1.1.0 rechnet Hash und Signatur jeder Fassung nach und gleicht jede in der Kette attestierte Spec-Bindung eines Runs gegen die gelieferten Fassungen ab; eine fehlende, manipulierte oder im Hash abweichende Fassung ist eine Verletzung. Fassungen anderer Mandanten liefert der Export nicht — eine Bindung ohne lieferbare Fassung fällt im Verifier auf. Die Erweiterung ist additiv und in beide Richtungen getestet: Der neue Verifier besteht Exporte 2.0 weiter (Spec-Bindungen dort als „nicht prüfbar“ ausgewiesen), der bisherige akzeptiert Exporte 2.1, weil er unbekannte Felder ignoriert; die CI baut ihn dafür eigens aus dem Vorgänger-Release.[15] Belegt ist damit, dass gegen genau diese Fassung gebaut wurde und dass sie unverändert vorliegt — nicht, dass sie fachlich richtig war. Ohne Freischaltung der Governance-Stufe (bei aktivem Lizenzsystem: unterhalb von Enterprise) bleibt der Export bei Schema 2.0.

Lieferkette und Wissensbestand

Der Supply-Chain-Nachweis aus Kapitel 14 des Whitepapers 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**

[14] Modul `verifier/` der SoftwareFabrik; Export-Schema 2.0 (seit 0.39.0: 2.1) in `AuditExportService`.
[15] Nachtrag zu ADR-0023 (Schema 2.1); Test `OfflineVerifierKompatibilitaetTest`.

Quelle, mehrere Projektionen statt einer gepflegten Kopie je Werkzeug; welche Guardrails-Version gewirkt hat, wird attestiert.

Gedächtnis ohne Vektorspeicher

Das Fünf-Schichten-Memory-Modell aus Kapitel 8 des Whitepapers 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 des Whitepapers 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, optional ephemere Container für Build und Scans je Schritt und optional der MCP-Gateway als eigener Prozess (19.4). Der Lizenzdienst (mit eigener Datenbank) ist getrennt und optional; er kann außerhalb der Unternehmensgrenze stehen oder ganz entfallen — standardmäßig ist das Lizenzsystem ausgeschaltet.

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 Lizenz ist ein mit Ed25519 signiertes Token, das offline gegen einen mit der Software ausgelieferten öffentlichen Schlüssel geprüft wird; seine Laufzeit hängt von Stufe und Betriebsform ab (30 Tage bis zu einem Jahr für Air-Gap-Installationen). Kurzzeitige Nichterreichbarkeit des Lizenzservers überbrückt ein Kulanzzeitraum; ist alles abgelaufen, startet kein neuer Lauf mehr — *fail closed* für die Ausführung, nicht für den Zugriff auf vorhandene Daten. Für Air-Gap-Installationen schaltet ein Feld im signierten Token die Erneuerung ganz ab; dann zählt nur das Ablaufdatum, und es gibt keinen zweiten Codepfad. Für die Air-Gap-Frage ist das die entscheidende Eigenschaft: Ein System, das für Verschlusssachen taugen soll, darf keine Online-Aktivierung als Betriebsvoraussetzung haben. (Die Fassungen bis 3.0 nannten hier noch einen OIDC-Provider und Lease-Token mit sieben Tagen Gültigkeit — ein Stand, den die Fabrik schon im April 2026, vor Release 0.2.0, durch das heutige Format abgelöst hatte.)[16]

Damit sind alle Bausteine für den vollständig getrennten Betrieb vorhanden: lokales Modell als Adapter, Agenten ohne Zugriff auf nicht freigegebene Werkzeugserver (19.4), Lizenz ohne Rückkanal, PR-Poller standardmäßig deaktiviert (unbeaufsichtigte periodische Remote-Kontakte sind eine bewusste Betriebsentscheidung), Host-Allowlist für Remotes.

[16] `LicenseManager` und `Ed25519JwsVerifier` im `license`-Slice; Laufzeiten im Lizenzdienst.

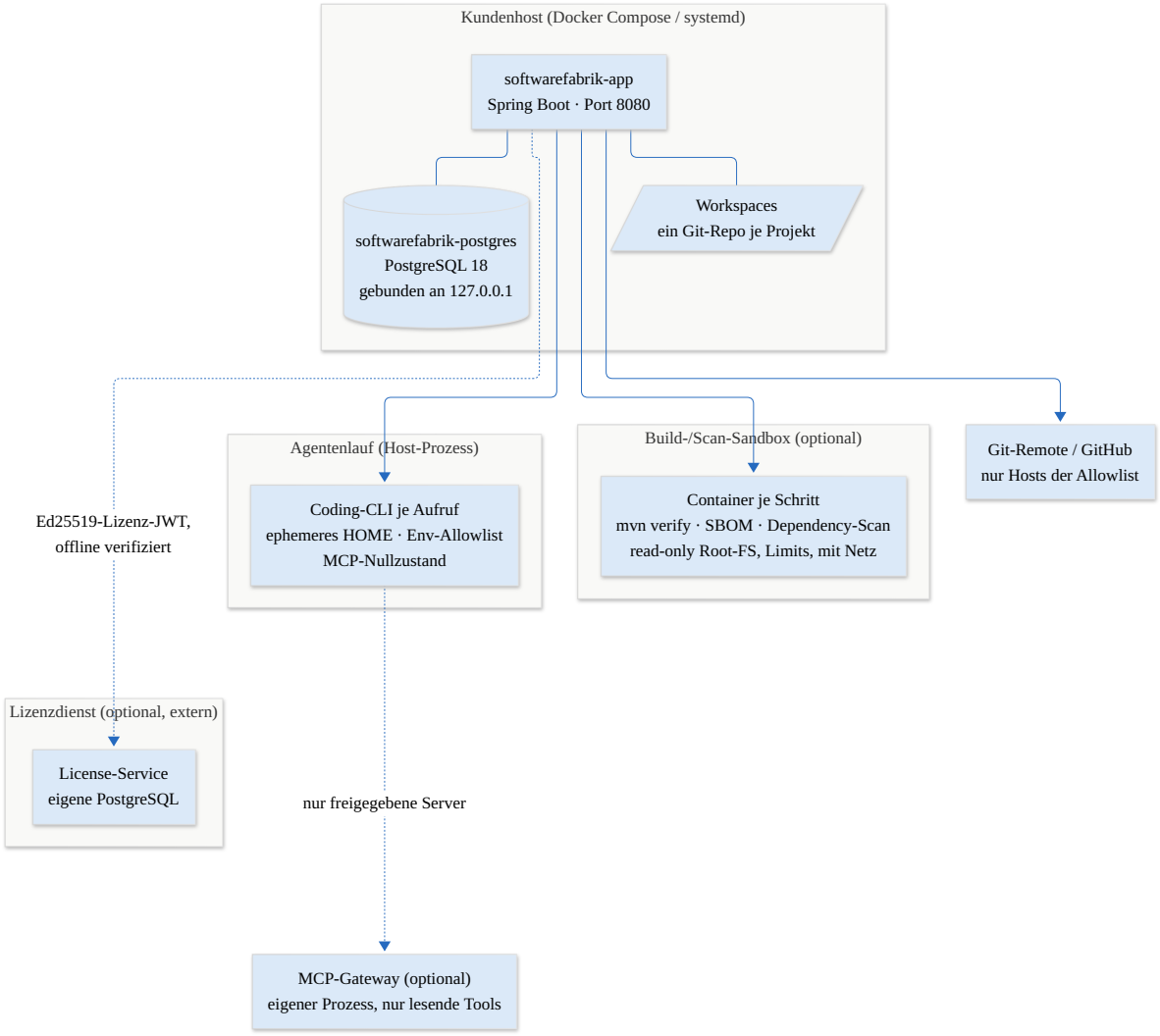


Abbildung 19.10: Deployment-Sicht: ein Anwendungscontainer, eine Datenbank, optional getrennter Lizenz-Stack. Air-Gap-fähig

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 des Whitepapers sowie ADR-1 in Kapitel 20 des Whitepapers) 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 Vielstimmigkeit deshalb von der Erzeugung auf die Bewertung — gemeint ist die Zahl unabhängiger Prüfer auf demselben Stand, nicht deren gleichzeitige Ausführung: Technisch laufen die Reviewer nacheinander, was frühere Fassungen dieses Whitepapers ungenau als „parallel“ beschrieben haben. 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 des Whitepapers). 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 des Whitepapers). 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 Lizenzdienst ohne Rückkanal, lokales Modell, keine Werkzeugserver ohne Freigabe — 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 20. Oktober 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
Coding-Agenten laufen in keiner Variante im Container, sondern als Host-Prozess (frisches Home je Aufruf, Umgebungs-Allowlist, MCP-Nullzustand); die Container-Sandbox kapselt nur Build und Scans und ist nicht der Default (19.4)	Einschränkung — Agent-Container als eigenes Arbeitspaket offen
MCP-Gateway nur mit lesenden Werkzeugen und nur für die Ausführung mit Claude Code; der Nachweisfall „Gateway projiziert“ und die Wirkung der Hook-Abschaltung sind noch nicht durch den Canary belegt	Verifikationslücke
Architektur-Altschuld: eingefrorene Modulzyklen und 13 direkte Repository-Zugriffe der Web-Schicht	dokumentierte, gezählte Schuld
Preisstatus unbekannter Modelle (heute 0 €, nötig wären `UNKNOWN`/Sicherheitsersatzwert); Abo-Läufe sind seit 0.33.0 als Schätzung gekennzeichnet	Budget- und Abrechnungslücke
Verifikationsstatus unsignierter Legacy-Auditdaten in der Plattform (undifferenziertes Ergebnis; der Offline-Verifier unterscheidet für Exporte bereits)	Nachweislücke
Policyabhängiges Verhalten bei fehlendem SBOM-Scanner (heute stets übersprungen)	Fail-Closed-Lücke

Punkt	Art
Spec-Driven Development: Die Abdeckungsprüfung ist eine Heuristik über Test-Referenzen — das gilt auch für die seit 0.39.0 per Policy erzwingbare Mindestabdeckung; Wirkung auf die Ergebnisqualität nicht gemessen (19.11)	Einschränkung

Gegenüber der Fassung 3.1 sind zwei Zeilen entfallen. Die SDD-Governance-Stufe ist mit 0.39.0 umgesetzt (19.11). Und der dort als „Status ungeklärt“ geführte Test-Isolationsdefekt — ein Test, der unter bestimmten Bedingungen in das reale Repository committete — ist geklärt: Er war bereits im Release 0.19.0 behoben, auf dessen Stand er als offen erhoben worden war. Ursache war ein Produktfehler, den die Tests sichtbar machten: Das Meilenstein-Release prüfte nur, ob der Projekt-Workspace existiert, nicht, ob er ein eigenes Git-Repository ist. Ohne eigenes liefen Commit und Tag gegen das nächste übergeordnete Repository — lag der Test-Workspace im Build-Verzeichnis der Fabrik, traf es das Repository der Fabrik selbst. Seither verlangt das Release ein eigenes Repository im Workspace, ein Regressionstest hält das fest (am Release-Tag 0.39.0 unverändert vorhanden).[17] Die Fallstudie hatte den Defekt aus einer Arbeitsnotiz übernommen, die nach der Behebung nicht nachgezogen wurde — auch das ein Befund über Nachweise. Nicht zu verwechseln mit zwei späteren, ebenfalls behobenen Kopplungen der Testsuite über die gemeinsame Testdatenbank (0.24.0: Kapazitätsgrenze; 0.36.0: unterschiedliche Signaturschlüssel je Testkontext, behoben durch einen Schlüssel je JVM).

Bereits mit Fassung 3.1 entfallen: Die seit 0.20.0 befristet akzeptierte Schwachstelle in einer Laufzeitabhängigkeit verschwand mit 0.31.0 durch ein Update der Abhängigkeit; der Abhängigkeits-Scan läuft seither ohne Ausnahmeeintrag. Der Image-Scan führt weiterhin begründete Ausnahmen für Hilfsprogramme des Basis-Images, die die Anwendung nie ausführt (17 Einträge, drei davon mit 0.33.0 neu).

Dazu vier Beobachtungen aus dem Entwicklungsverlauf (47 Releases in rund sechs 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.

[17] `MilestoneReleaseService.gibFrei`; Test `MilestoneReleaseServiceTest`.

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 des Whitepapers) — 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. Dass die Fehlerklasse damit nicht erledigt war, zeigten die Releases 0.33.0 und 0.34.0: LLM-Reviewer, die einen Fehler als bestandenes Review werteten, ein Gate auf leerem Diff (19.5), Läufe, die mit 0 € bewertet wurden (19.4), und eine Containervariante, deren Startmeldung eine Agenten-Isolation nahelegte, die es nicht gab (19.4). Alle vier Fälle sahen in der Oberfläche unauffällig aus. Ein weiterer Fall derselben Familie fiel erst mit 0.39.0 auf: Policy-Ablehnungen vor der Anlage eines Laufs, die der Code attestierte, die aber nie in der Kette ankamen, weil sie mit der Ablehnung zurückgerollt wurden (19.3). Eine leere Liste von Ablehnungen ist von „nichts abgelehnt“ nicht zu unterscheiden.

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 des Whitepapers 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 0.39.0

Diese Tabelle ist die autoritative Zuordnung von Releases zu Roadmap-Stufen und Entwicklungslinien; alle anderen Stellen des Whitepapers — Management Summary, Praxis-Checks, Kennzahlen-Erhebungsvermerk (19.1) und die ADR-Statuszeilen — verweisen hierher, statt den Verlauf zu wiederholen. Ab 0.31.0 gehören die Releases nicht mehr zur Workflow-Roadmap dieses Abschnitts, sondern zu zwei neuen Linien: der Absicherung und dem unabhängigen Nachweis der Agentenläufe (19.4, 19.6) und Spec-Driven Development (19.11).

Release	Stufe / Linie	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)
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

Release	Stufe / Linie	Inhalt	Migrationen
0.31.0	Absicherung	MCP-Nullzustand für alle Agentenläufe, Canary-Isolationsnachweis je CLI-Version, Plugins standardmäßig gesperrt; Spring Boot 4.1	V56
0.32.0	Nachweis	Offline-Verifier und Audit-Export 2.0: Prüfung ohne Zugriff auf die Fabrik, Vertrauensanker beim Prüfer	keine
0.33.0	Absicherung	Claude-Anbindung: vier Stellen „Erfolg ohne Beleg“ behoben; Abo-Modus als persönliche Nutzung, Abo-Kosten als Schätzung gekennzeichnet (0.33.1: Wartung)	V58
0.34.0	Absicherung	MCP-Gateway (opt-in, nur lesende Werkzeuge, jeder Aufruf attestiert); Hooks importierter Repositories laufen bei Claude Code nicht mehr mit; Rollen- und Mandantenprüfung der Workflow-Ebene; korrekte Kosten der Claude-Läufe; Aufwandsfeld an der Planfreigabe	V59–V60
0.35.0	SDD	Produkt-Spec mit Fassungen, Freigabe und Akzeptanzkriterien-IDs; Kriterien erreichen Backlog und Quality Gate	V61–V62
0.36.0	SDD	Traceability bis zur Kriteriums-ID: Commit-Trailer, attestierte Spec-Bindung, Workflow-Tasks gegen dieselbe Fassung	V63
0.38.0	SDD	Conformance-Reviewer, Spec-Drift, Spec-Run mit Rückfragen, Reverse-Spec; bündelt die Roadmap-Stufen 0.37 und 0.38 (eine Version 0.37.0 gibt es nicht)	V64–V65
0.39.0	SDD	Governance (bei aktivem Lizenzsystem nur Enterprise): Spec-Freigabe und Mindest-Kriterienabdeckung als Policy-Pflicht, in zwei Compliance-Profilen gesetzt; Audit-Export 2.1 mit signierten Spec-Fassungen, Offline-Verifier 1.1.0; SDD-Kennzahlen; Policy-Ablehnungen vor der Run-Anlage erstmals dauerhaft in der Audit-Kette	keine

Stand: 20. Oktober 2026, erhoben auf den Release-Tags. Die Details der Workflow-Stufen nennt der Kasten zum Umsetzungsstand weiter unten, die der Linien Absicherung und Nachweis die Abschnitte 19.4 bis 19.6, die der SDD-Linie Abschnitt 19.11.

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 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

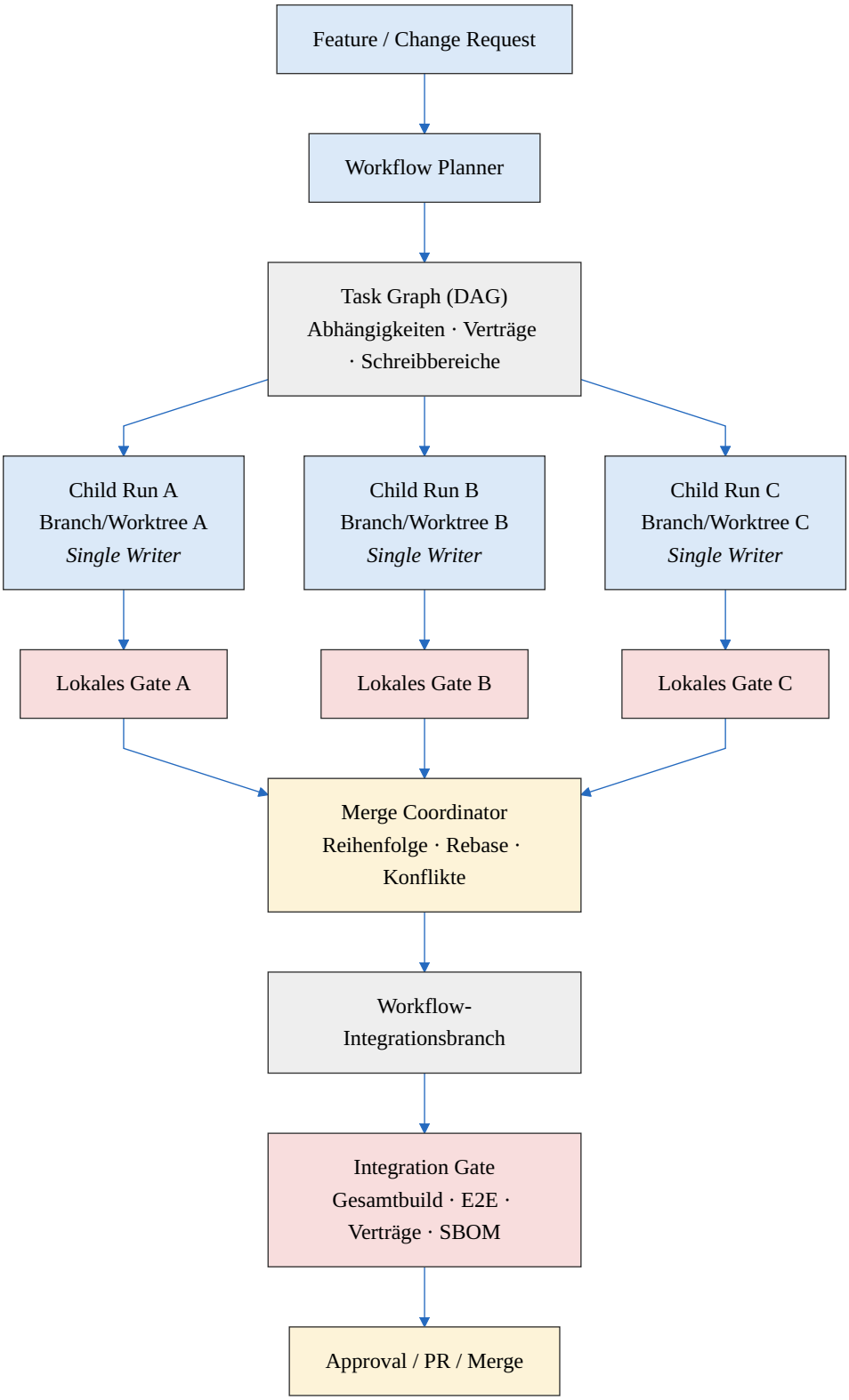


Abbildung 19.11: Architektur der parallelen Agenten-Workflows: isolierte Child Runs unter einem Parent Workflow, zusammengeführt über Merge Coordinator und Integration Gate. Bis Release 0.30.0 umgesetzt, hinter deaktiviertem Feature-Flag

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)
4	Dynamisches Replanning und Merge Intelligence: versionierte Planänderungen, Konfliktklassifikation, Rebase-/Revalidierungs-Pipeline, Eskalation mit vollständigem Kontext	hoch — **umgesetzt**

Stufe	Inhalt	Risiko
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, mit Nachträgen bis 0.39.0, Stand 20. Oktober 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 gleichzeitige 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 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 des Whitepapers) 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 — und seit 0.34.0 auch an der Planfreigabe des Workflows (Migration V59). 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.

Ein Nachtrag aus 0.34.0 bestätigt die Vorsicht dieser Stufe: Die Kennzahl *Eingriffe* zählte bis dahin nur Freigaben zu Child Runs, nicht die Planfreigabe — sie untertrieb also systematisch um mindestens eins je Workflow, und zwar ausgerechnet um die Entscheidung, ohne die kein Agent startet. Seither zählen Plan- und Run-Freigaben gemeinsam, ohne Doppelzählung. Die Zahl bleibt eine ausgewiesene Untergrenze, weil der Workflow nur die jeweils letzte Planfreigabe festhält. Im selben Release erhielt die Workflow-Ebene ihre Rollen- und Mandantenprüfung (19.6); seit 0.36.0 und 0.38.0 binden Workflow-Tasks zudem eine freigegebene Spec-Fassung und werden bei einer Spec-Änderung wie bei einer Vertragsänderung auf Neuplanung gesetzt (19.11); seit 0.39.0 kann eine Policy diese Bindung für jeden schreibenden Task zur Bedingung der Planfreigabe machen.

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 des Whitepapers in belegte Aussagen verwandeln (vgl. Kapitel 15.6 des Whitepapers).

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.

Die Anforderungsebene aus 19.11 setzt diese Linie fort: Parallele Child Runs arbeiten seit 0.36.0 nicht nur gegen dieselben Verträge, sondern gegen dieselbe freigegebene Anforderungsfassung.

19.11 Spec-Driven Development: die Anforderungsebene *(seit 0.35.0)*

Dieser Abschnitt beschreibt die jüngste Entwicklungslinie der Fabrik, **Spec-Driven Development (SDD)**: Anforderungen werden zu versionierten, freigegebenen Artefakten, gegen die Läufe gebunden und geprüft werden.

> **Status:** Implementiert in den Releases 0.35.0, 0.36.0, 0.38.0 und 0.39.0 (Tabelle in 19.10) — opt-in über ein eigenes Feature-Flag (Standard aus) und, bei aktivem Lizenzsystem, nur in den Lizenzstufen Professional und Enterprise; die Governance-Stufe aus 0.39.0 (Spec-Pflicht per Policy, Spec-Fassungen im Audit-Export, SDD-Kennzahlen) nur in Enterprise. Je Projekt gilt zusätzlich ein **Spec-Modus**: `AUS` (Standard), `LEICHT` oder `STRENG`. Ohne Flag verhält sich die Fabrik wie vorher; im Modus `AUS` laufen Build- und Plan-Runs unverändert, nur ein ausdrücklich gestarteter Spec-Run ist auch dort möglich. Die zugrunde liegende Architekturentscheidung führt die Fabrik noch als *vorgeschlagen*, nicht als angenommen. Mit 0.39.0 ist die Roadmap dieser Linie vollständig umgesetzt. Erfahrungen aus dem Betrieb oder Messungen zur Wirkung liegen noch nicht vor.

Die Lücke: gut ist nicht dasselbe wie richtig

Vor 0.38.0 prüfte das Quality Gate, ob Code *gut* ist — Sicherheit, Architektur, Abhängigkeiten, Behauptungen des Agenten (19.5) —, aber nicht, ob er das *Richtige* tut. Die Fabrik hatte viele Bausteine, die spezifikationsgetriebener Entwicklung ähneln: Projekt-Artefakte aus dem Wizard, Plan-Runs mit Backlog, versionierte Verträge, Freigaben, Warum-Trace. Es fehlte eine explizite, versionierte und abnehmbare Anforderungsebene. Akzeptanzkriterien existierten nur als Freitext in Plänen und erreichten das Gate nie (19.5); Freigaben hingen an Lauf und Phase, nicht an

Anforderungen; und parallele Child Runs bekamen nur ihren Task-Auftrag, keinen gemeinsamen Anforderungsstand. Das Leitbild der Fabrik — *beschreiben, bauen lassen, genehmigen* — hatte damit für den ersten Schritt kein prüfbares Artefakt.

Die Produkt-Spec

Die Antwort ist die **Produkt-Spec** als einzige Quelle für Anforderungen: Markdown mit einem kurzen Kopf (ID, Version, Status, Verantwortlicher, verknüpfte Verträge) und drei Pflichtabschnitten — Ziel, Nicht-Ziele, Akzeptanzkriterien; Randfälle und offene Fragen sind optional. Jedes Akzeptanzkriterium (englisch *acceptance criterion*, daher das Kürzel AC) trägt eine Pflicht-ID nach dem Schema `AC-<nnn>-<nn>`, eindeutig innerhalb der Spec und stabil über Fassungen; eine entfernte ID wird nicht wiederverwendet. Empfohlen ist die Formulierung in **EARS** (*Easy Approach to Requirements Syntax*), einer kleinen Menge von fünf Satzschablonen — allgemein gültig, ereignisgesteuert, zustandsgesteuert, unerwünschtes Verhalten, optionale Funktion [2]; Given/When/Then ist erlaubt. Ein Kriterium aus der ersten Spec der Fabrik lautet etwa:

> **AC-001-08** — Wenn ein Agent im Workspace eine Datei unter `specs/` ändert, dann soll das System die Änderung als neue DRAFT-Version übernehmen und niemals eine Version dadurch freigeben.

Der Parser prüft ID-Schema, Eindeutigkeit und Pflichtabschnitte, bewusst nicht die Satzform — präzise Anforderungen ohne Formalismus-Hürde.[18]

Die **Datenbank ist führend**, die Datei im Workspace (`specs/SPEC-<nnn>-<slug>/spec.md`) ist Projektion der freigegebenen Fassung; ein neuerer Entwurf liegt daneben. Jede Fassung ist unveränderlich und trägt einen SHA-256-Hash. Ändert ein Agent eine Datei unter `specs/`, übernimmt die Fabrik das als neue Entwurfsfassung — nie als Freigabe. Die Aufteilung folgt demselben Muster wie bei den Verträgen der Workflow-Ebene und aus demselben Grund: Unveränderlichkeit, Mandantentrennung und die Signierbarkeit (seit 0.39.0 im Audit-Export) sichert die Datenbank, der gewohnte Arbeitsfluss von Menschen und Agenten findet in Dateien statt.

Architektonisch ist `spec` ein eigener Slice, der nur das Querschnittspaket `common` und die Audit-Kette kennt. Ein reiner Blatt-Slice (19.2) schied aus, weil Orchestrierung und Gate die Spec konsumieren müssen; der schmale Importrand erlaubt das, ohne einen neuen Modulzyklus zu erzeugen.

Freigabe als eigenes Gate

Eine Spec durchläuft `DRAFT` → `APPROVED` → `SUPERSEDED`. Freigeben darf ab der Rolle Maintainer; die Freigabe validiert die Pflichtabschnitte und IDs, löst die vorher freigegebene Fassung ab, hält die Historie fest und schreibt `SPEC_APPROVED` mit Spec-ID, Version und Hash in die Audit-Kette. Seit 0.36.0 ist sie gegen Gleichzeitigkeit gesichert: Zwei parallele Freigaben hinterlassen genau eine freigegebene Fassung. Damit gibt es neben der Freigabe eines Laufs (19.3) erstmals die Freigabe einer **Anforderung** — eine Entscheidung mit Hebelwirkung, weil alle späteren

[18] Slice `spec` mit `SpecParser`, `SpecVersion`, `SpecService`; Vorlage `specs/_vorlage/spec.md` im Repository der Fabrik.

Läufe mit Spec-Bezug gegen sie gebunden und geprüft werden (Human Authority, Prinzip AP-6 aus Kapitel 2 des Whitepapers).

Der Spec-Modus steuert, wie viel davon verbindlich ist. In `LEICHT` sind Specs optional und alle Conformance-Befunde beratend. In `STRENG` lehnt die Fabrik die Aktivierung eines Backlog-Elements ab, dessen Spec keine freigegebene Fassung hat oder das unbekannte Kriteriums-IDs nennt; die Planfreigabe eines Workflows scheitert, wenn ein schreibender Task keinen Bezug auf eine freigegebene Fassung hat; und der Conformance-Reviewer (unten) wird zum blockierenden Pflicht-Reviewer. Der Weg in die Verbindlichkeit folgt damit derselben Regel wie beim Gate selbst (19.5): erst messen, dann durchsetzen.

Traceability über IDs, nicht über Texte

Die Kriteriums-IDs ziehen sich durch den ganzen Ablauf (0.35.0 und 0.36.0). Im Spec-Modus verlangt die Plan-Direktive je Plan die Zeilen `Spec:` und `AC:`; die Fabrik liest beide beim Indexieren und speichert sie am Backlog-Element. Ein Build-Run, der ein solches Element umsetzt, übergibt allen Reviewern die Kriterien mit ID und Text — das bis dahin fest leere Feld aus 19.5. Beim Start attestiert `RUN_SPEC_GEBUNDEN` Spec, Fassung, Hash und Kriterien in der Audit-Kette; der Warum-Trace zeigt sie an. Fabrik-Commit und Pull Request tragen die Trailer `Spec:` und `Implements:`, auch wenn der Agent selbst keine gesetzt hat. Den Agenten erreicht der Kontext werkzeugneutral: als Kopf des Arbeitsauftrags mit den gebundenen Kriterien, direkt hinter den Guardrails (deren Inhalt und Hash bleiben unverändert), und als Spec-Index in der projizierten `AGENTS.md` (19.6) — in Tests mit nachgebildeten CLIs für Claude, Codex und Gemini geprüft (belegt ist die Übergabe, nicht die Befolgung durch das Modell).

Auf der Workflow-Ebene deklariert ein Task Spec und Kriterien; gebunden wird beim Start des Child Runs, nicht bei der Planung — dieselbe Regel wie bei den Verträgen (19.10, Stufe 3). Alle Child Runs eines Workflows arbeiten gegen dieselbe Fassung, auch wenn die Spec zwischenzeitlich neu freigegeben wurde. Und um kein zweites Modell zu erzeugen, veröffentlicht die Freigabe den Kriterienblock einer Spec zusätzlich als Vertrag in der Contract Registry; Bindung und Veraltet-Markierung der Workflow-Ebene folgen demselben Muster wie bei den Verträgen (eigene Bindung am Task, Migration V63). Die Spec bleibt die einzige Stelle, an der Kriterien bearbeitet werden.

Conformance: prüfen gegen die Spec

Mit 0.38.0 prüft das Gate erstmals regelbasiert gegen die Anforderungen. Der dafür neue Reviewer arbeitet ohne Agenten-CLI und springt nur bei Läufen mit Spec-Bezug an (Name: `spec-conformance`).[19] Er prüft dreierlei: ob jedes gebundene Kriterium in mindestens einer Testdatei des Repositorys referenziert wird (Testname, Dateiname, Annotation oder Kommentar mit der ID) — fehlt die Referenz, entsteht ein Befund mit der Kriteriums-ID; ob Tests IDs nennen, die es in der gebundenen Fassung nicht gibt (Hinweis); und ob geänderte Produktivdateien keinem Kriterium zugeordnet sind, weder über einen `Implements:`-Trailer noch über einen Pfad-Hinweis der Spec (Hinweis „ungedecktes Verhalten“ mit begrenzter Pfadliste). Die Erkennungsmuster

[19] `SpecConformanceReviewerAdapter` im `review`-Slice; der Spec-Kontext erreicht ihn als Metadatum der Prüfanfrage, sodass `review` den `spec`-Slice nicht importiert.

sind konfigurierbar, der Scan ist begrenzt. In `LEICHT` sind alle Befunde beratend; in `STRENG` ist eine fehlende Abdeckung `CRITICAL`, und das Pipeline-Gate läuft für diesen Lauf blockierend — auch wenn der globale Gate-Modus milder eingestellt ist, mit einem Hinweis im Lauf-Protokoll.

Ehrliche Einordnung: Die Abdeckungsprüfung ist eine Heuristik. Eine Kriteriums-ID im Namen eines Tests belegt eine *Referenz*, nicht, dass der Test das Kriterium tatsächlich prüft — dieselbe Konstruktvaliditätsgrenze, die 19.9 für Testabdeckung und bestandene Gates benennt. Die Fabrik selbst führt Falsch-Positive als Restrisiko; beratend ist der Reviewer deshalb im Modus `LEICHT`, blockierend nur, wenn ein Projekt ausdrücklich `STRENG` wählt. Was die Prüfung leistet, ist Sichtbarkeit: Ein Kriterium ohne jeden Testbezug fällt auf, und Verhalten ohne Anforderung auch. Der Zuschnitt ist derselbe wie beim Halluzinations-Reviewer (19.5) — verglichen werden Behauptung und Beleg, hier die Anforderung und ihr Testbezug.

Spec-Drift: was eine Änderung veralten lässt

Als **Spec-Drift** bezeichnet die Fabrik die Abweichung zwischen einer neu freigegebenen Spec-Fassung und der Arbeit, die gegen die vorherige geplant wurde. Ändert sich eine freigegebene Spec, ist die Frage, welche geplante oder laufende Arbeit dadurch auf einer überholten Grundlage steht. Die Freigabe einer neuen Fassung vergleicht sie deshalb auf Kriteriumsebene mit der vorher freigegebenen — neu, entfernt, geändert — und schreibt das Ergebnis als `SPEC_DRIFT` in die Audit-Kette. Offene Backlog-Elemente, die an ein entferntes oder geändertes Kriterium gebunden sind, erhalten eine Begründung und erscheinen als *veraltet*; ein Element ohne Kriteriums-IDs gilt dabei für alle Kriterien der Spec — im Zweifel lieber eine Prüfung zu viel als ein stiller Bruch, dieselbe Abwägung wie bei der minimalen Normalisierung der Verträge (19.10, Stufe 3). Gebundene Workflow-Tasks gehen wie bei einer Vertragsänderung in die Neuplanung — eine geänderte Anforderung fällt damit unter den Grund *Vertragsänderung*, also unter die neuen Eingaben, die das Rücksetzen eines Tasks nach Stufe 4a erlauben. Die Kriterienverträge sind dafür strukturell vergleichbar geworden: Ein entferntes Kriterium gilt als Bruch, ein geänderter Text als Hinweis. Nennt eine Spec einen verknüpften Vertrag, dessen aktuelle Fassung einen Bruch trägt, zeigt die Spec-Übersicht sie als *prüfbedürftig* — abgeleitet beim Lesen, nicht gespeichert.

Spec-Run: Anforderungen erarbeiten statt nur verwalten

Vor 0.38.0 entstanden Specs nur von Hand. Der **Spec-Run** — die dritte Run-Art (19.3) — erarbeitet sie mit: aus einer Idee in Freitext oder zum Schärfen einer bestehenden Spec. Seine Direktive verlangt Ziel, Nicht-Ziele, Kriterien mit IDs nach der Vorlage, je Kriterium Negativ- und Grenzfälle und einen Abschnitt „Offene Fragen“. Übernommen wird ausschließlich, was unter `specs/` liegt — das Gegenstück zur Übernahme nur der Pläne beim Plan-Run; alles andere wird verworfen und im Protokoll genannt. Jede geänderte Spec-Datei wird eine neue Entwurfsfassung, auch wenn der Parser sie beanstandet; die Befunde stehen an der Fassung. Kein Build, kein Branch, kein Merge. Für die Modellwahl gibt es eine eigene Routing-Rolle, die wie die Planung ein starkes Modell anfordert.

Eine Entscheidung mit Verallgemeinerungswert betrifft **Rückfragen**. Allen anderen Läufen untersagen die generierten Anweisungen, Rückfragen zu stellen — ein Agent ohne Dialogpartner soll nicht warten. Für Spec-Runs, und nur für sie, wird diese Regel ersetzt: Fragen gehören in den

Abschnitt „Offene Fragen“. Die Fabrik liest sie beim Abschluss aus, legt sie einzeln ab und versetzt den Lauf in den neuen, nicht-terminalen Zustand `WAITING_FOR_INPUT` — der bewusst keinen Kapazitäts- und keinen Workspace-Platz belegt, denn es läuft kein Prozess, und ein Mensch kann Tage brauchen. Jede Frage wird in der Oberfläche beantwortet oder als nicht relevant markiert, beides endgültig und mit Akteur und Zeitpunkt attestiert. Fortgesetzt wird erst, wenn keine Frage mehr offen ist — und zwar als *neuer*, verknüpfter Spec-Run mit den Antworten als Eingabe, der wieder eine neue Entwurfsfassung erzeugt. Der Dialog wird so zu einer Folge nachvollziehbarer Läufe statt zu einem Gesprächsverlauf, den niemand zitieren kann — dieselbe Begründung wie beim Verzicht auf einen Vektorspeicher (19.6).

Für importierte Repositories gibt es die **Reverse-Spec**: Ein Spec-Run mit eigener Direktive leitet aus Code und Tests einen Entwurf ab, nennt je Kriterium die Quelle, führt Annahmen, Nicht-Ziele und offene Fragen ausdrücklich auf und soll nichts erfinden. Das adressiert den Fall, in dem die Fabrik nicht auf der grünen Wiese, sondern auf einem bestehenden System beginnt. Das Restrisiko benennt die Fabrik selbst: Generierte Specs können plausibel und falsch sein. Sie werden deshalb nie automatisch freigegeben; die Freigabe bleibt eine menschliche Entscheidung. Die Direktiven laufen werkzeugneutral über Auftrag und Anweisungsdatei — in Tests mit nachgebildeten Claude- und Codex-CLIs geprüft —, und der Mock-Adapter macht den Pfad ohne Vendor-CLI vorführbar.

Governance: Spec-Pflicht, Nachweis, Kennzahlen *(0.39.0)*

Die bisherigen Stufen machten die Anforderungsebene verfügbar; ob sie genutzt wird, entschied das Projekt über seinen Spec-Modus. Release 0.39.0 gibt diese Entscheidung für regulierte Umgebungen an die Policy-as-Code (19.6) — dorthin, wo auch die übrigen Durchsetzungsregeln liegen. Zwei neue Felder, beide in der kanonischen Form nur bei Abweichung vom Standard und deshalb ohne Bruch bestehender Policy-Signaturen:

- **Spec-Freigabe als Pflicht.** Verlangt eine wirksame Policy — global oder die eines Mandanten, wie bei den übrigen Regeln gilt die strengere — die Spec-Freigabe, startet ein Build-Run nur mit Bezug auf eine *freigegebene* Fassung, und die Planfreigabe eines Workflows verlangt diesen Bezug für jeden schreibenden Task. Die Policy schlägt dabei den Spec-Modus des Projekts: Auch im Modus `AUS` wird abgelehnt, mit einem lesbaren Grund (Spec-Modus aus, kein Spec-Bezug, keine freigegebene Fassung, SDD nicht freigeschaltet). Plan- und Spec-Runs sind nicht betroffen, Child Runs prüft bereits die Planfreigabe.
- **Mindestabdeckung der Kriterien.** Liegt der Anteil der gebundenen Kriterien, die der Conformance-Reviewer in Tests referenziert findet, unter dem Policy-Wert, blockiert das Gate — auch im beratenden Modus, nicht aber bei abgeschaltetem Gate — und die Korrekturschleife (19.3) beginnt. Ohne Spec-Bezug greift die Regel nicht; dafür ist die Pflicht oben zuständig. Liefert der Reviewer kein Ergebnis, gilt nichts als belegt (fail-closed, AP-7).

Die Ablehnungen stehen mit Grund in der Audit-Kette: beim Run-Start und im Gate als `RUN_POLICY_DENIED` (mit Zeitpunkt `start` beziehungsweise `quality-gate`), bei der Planfreigabe als neues Ereignis `WORKFLOW_PLAN_DENIED` — vorher wurde eine abgelehnte Planfreigabe gar

nicht protokolliert. Dauerhaft sind die Start-Ablehnungen erst mit diesem Release (19.3). Das Gate-Ergebnis trägt die Zahl abgedeckter und gebundener Kriterien. Die Compliance-Profile EU AI Act und BAIT/MaRisk/DORA setzen die Spec-Pflicht; eine Mindestabdeckung setzt keines der Profile (19.6).

Für den Nachweis nach außen enthält der Audit-Export ab Schema 2.1 die gebundenen Spec-Fassungen mit signiertem Hash, und der Offline-Verifier prüft sie unabhängig von der Fabrik (19.6). Damit lässt sich gegenüber einer Revision belegen, gegen welche abgenommene Anforderung gebaut wurde.

Hinzu kommen **SDD-Kennzahlen** je Projekt, für Einzel-Runs und je Workflow: der Anteil der Build-Starts mit freigegebener Spec, die Kriterienabdeckung am Ende eines Laufs, die Zahl der Spec-Drifts und die Spec-Änderungen je freigegebenem Meilenstein. Wie die übrige Kennzahlenmessung (19.10, Stufe 6) sind sie vollständig aus Audit-Kette und Meilensteinen abgeleitet, ohne eigene Tabelle und ohne Migration; leere Stichproben bleiben leer statt null, und als Messlücken stehen daneben die Abdeckung vor 0.39.0 (das Gate-Ergebnis trug sie noch nicht) und Releases außerhalb der Fabrik.

Die Governance-Stufe ist bei aktivem Lizenzsystem der Lizenzstufe Enterprise vorbehalten (ohne Lizenzsystem genügt das SDD-Flag); in Professional bleiben die Funktionen aus 0.35.0 bis 0.38.0 verfügbar, Spec-Pflicht und Mindestabdeckung sind dort wirkungslos, und die Oberfläche nennt den Grund. Architektonisch lesen `run` und `workflow` die neuen Regeln über einen schmalen Port im Querschnittspaket, ohne eine neue Kante zwischen den Slices.[20]

Einordnung

Kennzeichnend ist weniger die einzelne Funktion als der Zuschnitt: Spec-Driven Development erfindet in der Fabrik kaum neue Mechanik, sondern verbindet vorhandene — Freigabe, Bindung beim Start, Veraltet-Markierung, Audit-Kette, Warum-Trace — über eine Anforderungsebene mit stabilen IDs. IDs statt Texte machen Traceability prüfbar und Drift berechenbar. Die Fabrik beschreibt ihre eigenen SDD-Stufen in diesem Format (fünf Specs im eigenen Repository, alle — auch nach ihrer Umsetzung — im Status Entwurf). Auch die Governance-Stufe folgt dem Muster: Spec-Pflicht und Mindestabdeckung sind zwei Felder der bestehenden Policy, der Nachweis ist ein additives Feld im bestehenden Export, die Kennzahlen sind aus der bestehenden Kette abgeleitet.

Offen bleibt die wichtigste Frage: ob die Anforderungsebene die Ergebnisse agentischer Läufe messbar verbessert. Die Mechanik ist implementiert; belegt ist damit, dass Anforderungen versioniert, freigegeben, gebunden, gegen Tests referenziert und — wo eine Policy es verlangt — erzwungen werden können, nicht, dass dadurch weniger falsche Software entsteht. Die SDD-Kennzahlen machen die Nutzung sichtbar, nicht die Wirkung; das wäre eine Aufgabe für die Messung aus Kapitel 15.6 des Whitepapers.

[20] Port `common.SpecGovernance`, implementiert vom `PolicyEnforcementService`; Felder `pflichtSpecFreigabe` und `minAcAbdeckung` in `PolicyInhalt`.

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] Alistair Mavin u. a. „Easy Approach to Requirements Syntax (EARS)“. In: *Proceedings of the 17th IEEE International Requirements Engineering Conference (RE'09)*. 2009, S. 317–322. DOI: 10.1109/RE.2009.9.

[3] *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).

[4] *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).