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 3.0 · **Stand: 10. August 2026**

Inhaltsverzeichnis

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 19.10), 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 9. August 2026 auf dem Release-Stand 0.30.0), nicht aus Projektdokumentation oder Erinnerung. Wo eine Aussage im Code verankert ist, nennt eine Fußnote die konkrete Klasse; die Klassennamen dienen der präzisen Verortung — das Repository des Systems ist derzeit nicht öffentlich.

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

19.1 Von der Referenzarchitektur zur Implementierung

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

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

Der Kernablauf

Ein Vorhaben durchläuft die Fabrik in sechs Schritten:

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

Kennzahlen der Codebasis

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

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

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

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

Bewusste Nicht-Ziele

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

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

19.2 Systemkontext und Bausteinsicht

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

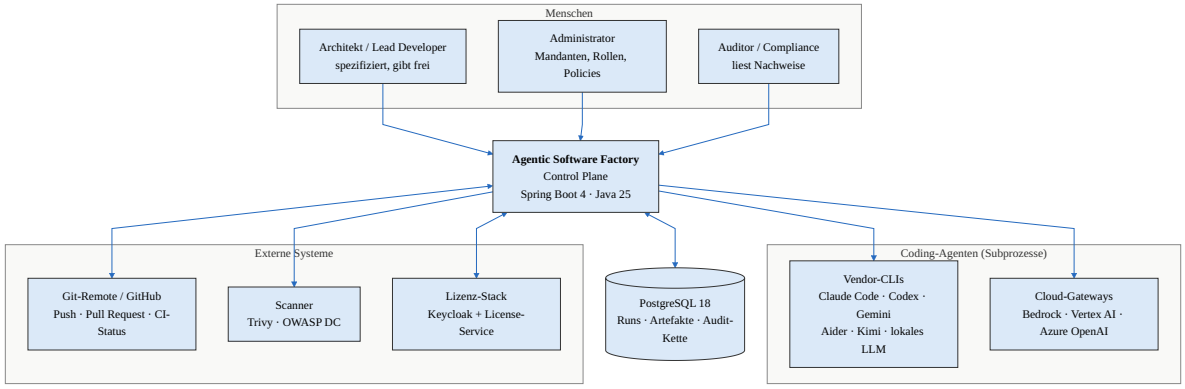

Abbildung 19.1: Systemkontext der Agentic Software Factory: eine Control Plane zwischen Mensch, Coding-Agenten und Zielrepository

Gegenüber dem Systemkontext aus Kapitel 3 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 30. Jeder Slice trägt seine eigenen Schichten (`domain`, `application`, `web`, teils `infrastructure`); externe Systeme — Coding-CLIs, Git, Maven, Scanner, GitHub-API — sitzen ausschließlich hinter Ports.

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

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

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

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

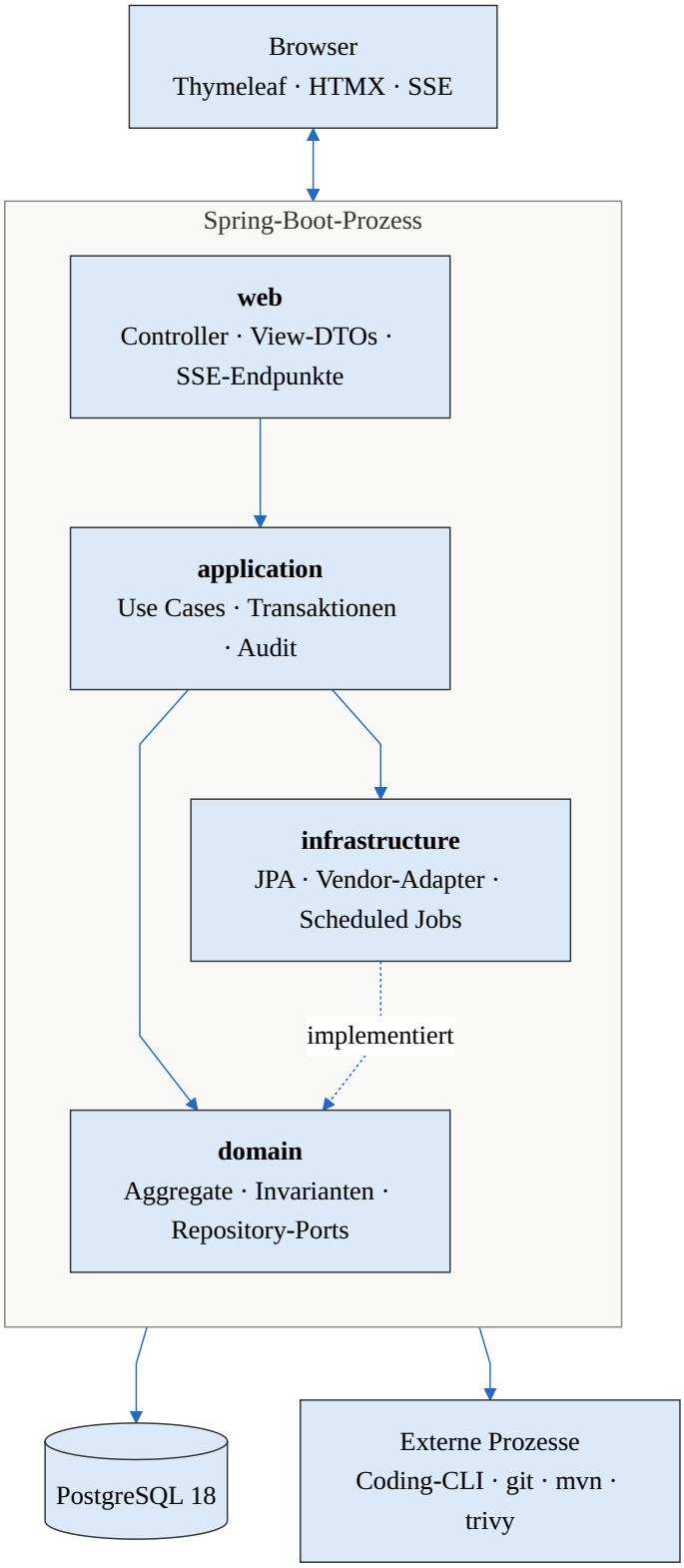

Abbildung 19.2: Bausteinsicht: modularer Monolith mit Ports-and-Adapters pro Slice; externe Werkzeuge ausschließlich hinter Ports

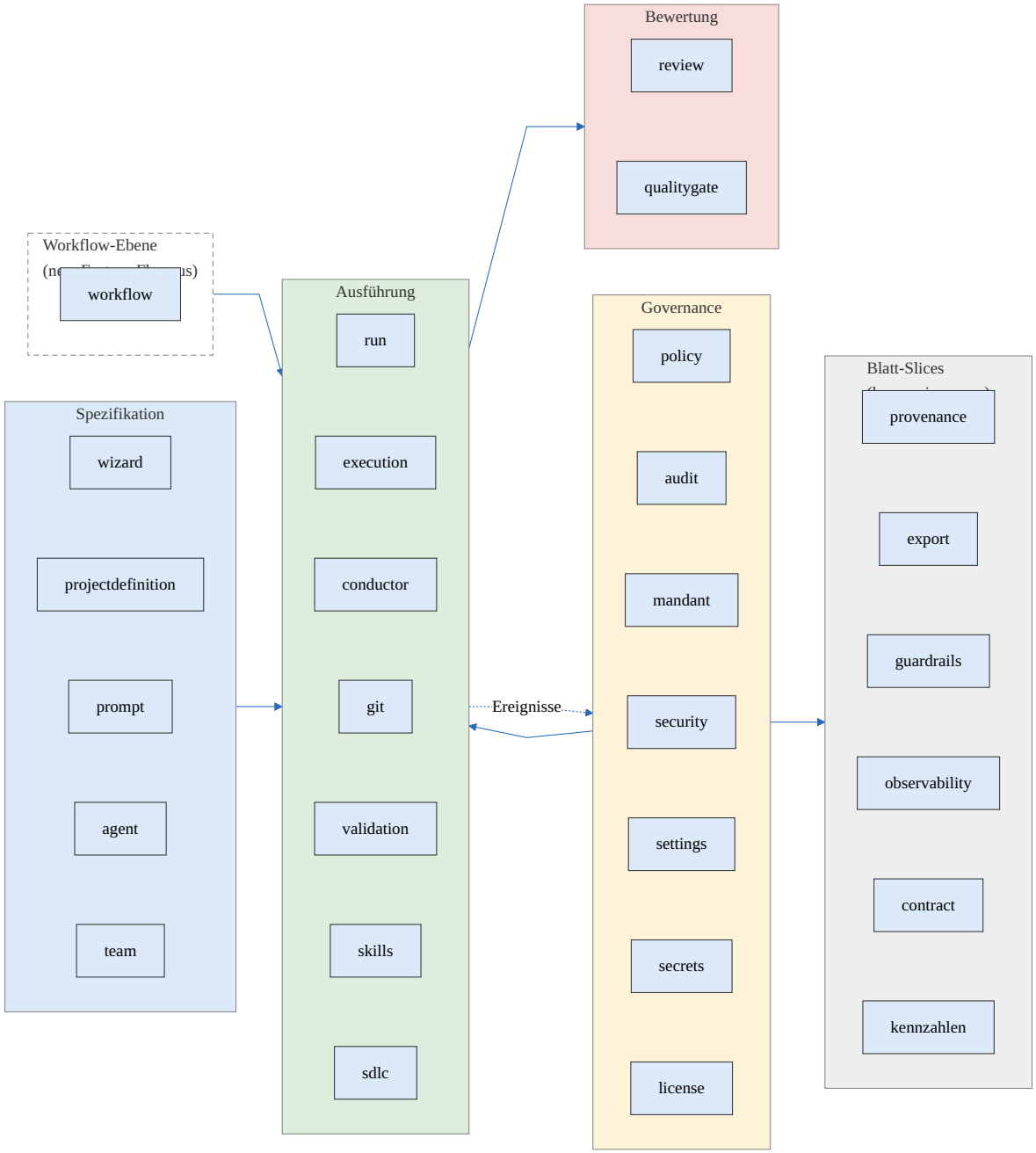

Abbildung 19.3: Fachliche Slices der Fabrik, gruppiert nach Aufgabe. Blatt-Slices (provenance, export, guardrails, observability, contract, kennzahlen) konsumieren nur; die technischen Querschnittspakete common und web sind nicht dargestellt

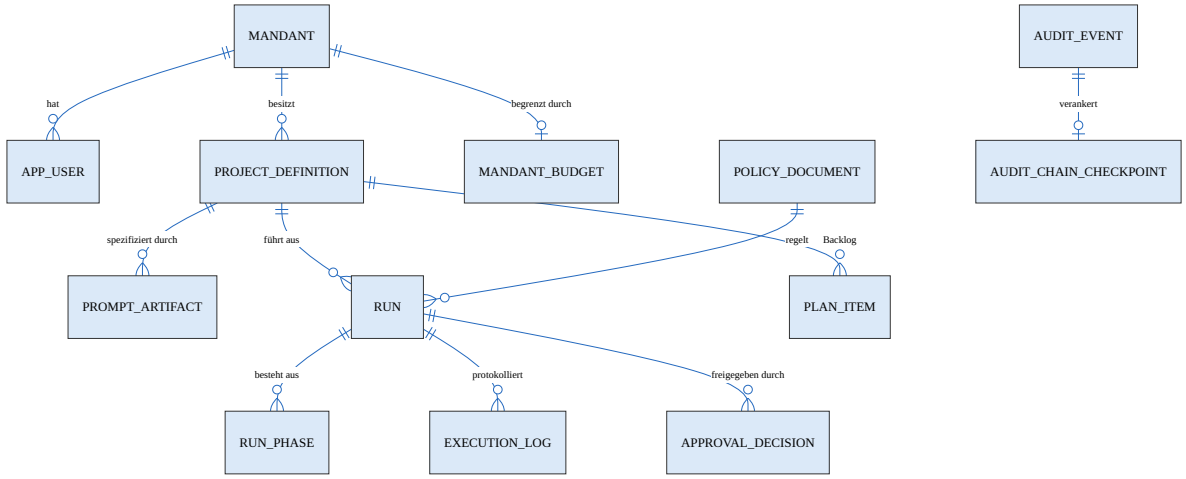

Abbildung 19.4: Kernaggregate der Run-Ebene; vollständig umfasst das Schema 58 Tabellen in 52 Flyway-Migrationen

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

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

Das Domänenmodell: der Entwicklungsprozess als Fachdomäne

Die Fabrik modelliert **den Entwicklungsprozess selbst** als Domäne. Nicht der generierte Code ist das Fachobjekt, sondern Spezifikation (`ProjectDefinition`, `PromptArtifact`), Lauf (`Run` mit `RunPhase`), Backlog (`PlanItem` mit Abhängigkeiten), Freigabe (`ApprovalDecision`), Regelwerk (`PolicyDocument`) und Nachweis (`AuditEvent`). Das ist die konsequente Anwendung von Kapitel 11 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 (Werk-

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

19.3 Das Ausführungsmodell: der Run

Der *Run* ist die Ausführungseinheit der Fabrik — das Gegenstück zum Execution Model aus Kapitel 7 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]

Zwei Run-Arten, sieben Phasen, 13 Zustände

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

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

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

Ablauf eines Build-Runs

Vier Stationen des Ablaufs sind konzeptionell bemerkenswert:

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

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

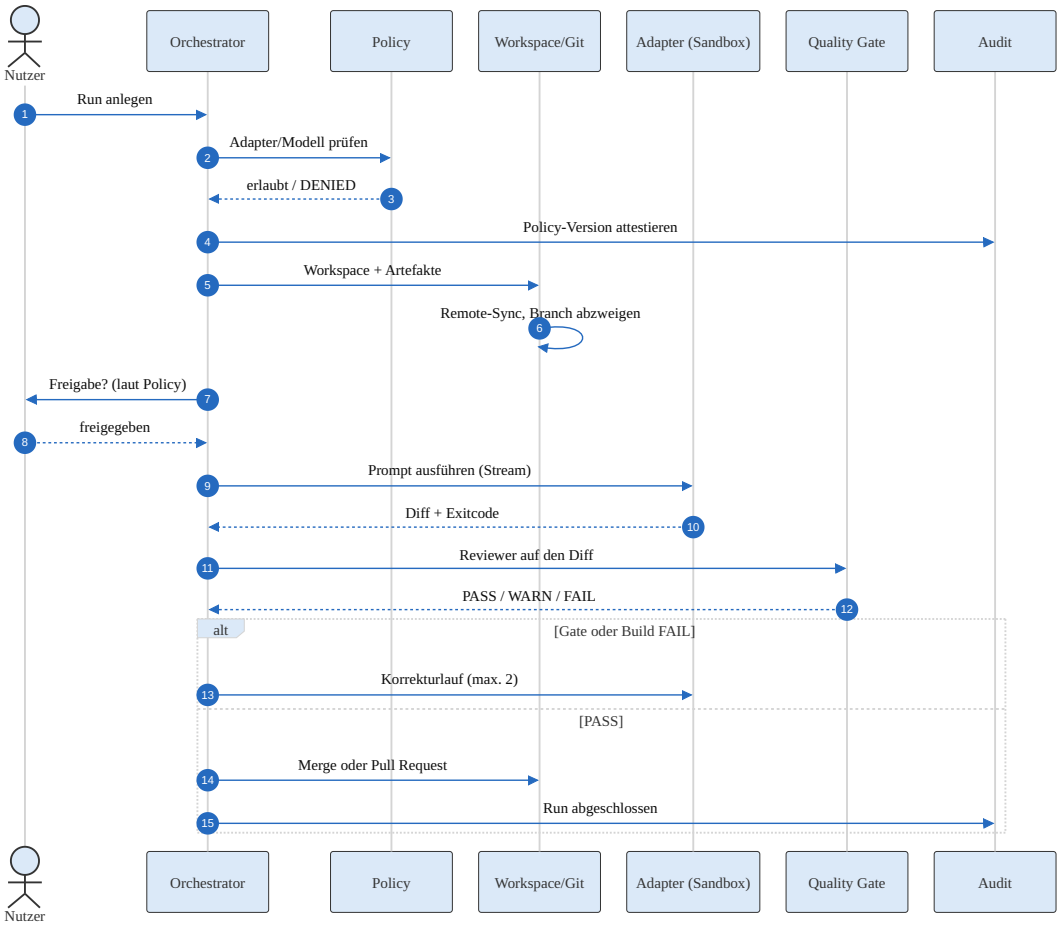

Abbildung 19.5: Laufzeitablauf eines Build-Runs von der Anlage bis zum Merge, inklusive Korrekturschleife und Approval-Punkten

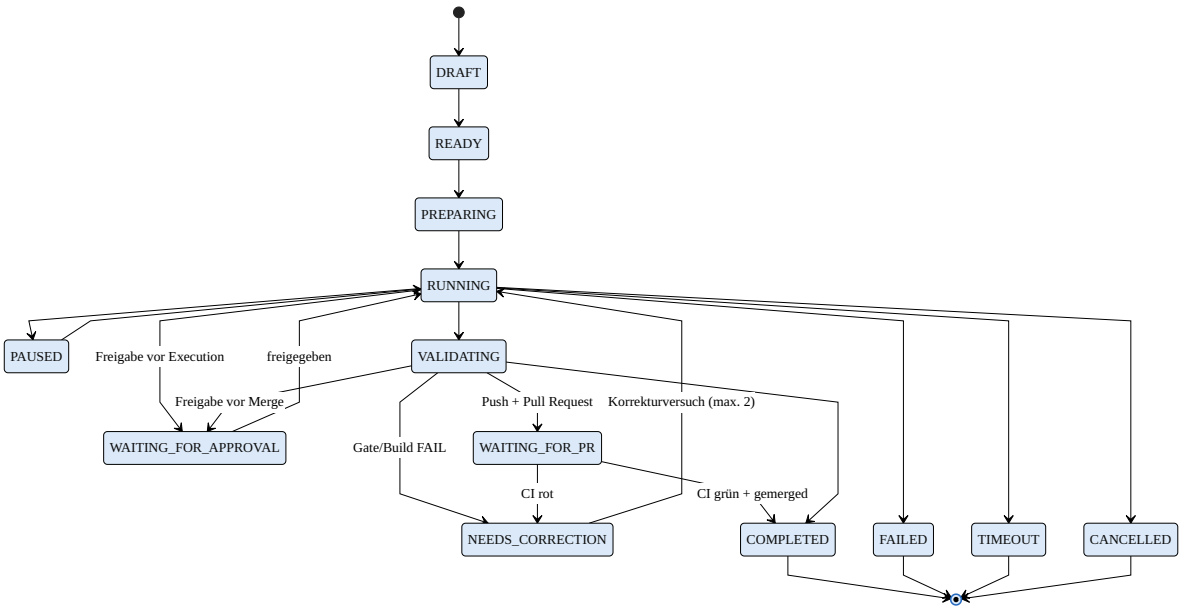

Abbildung 19.6: Zustandsmodell eines Runs. Alle Übergänge sind zentral hinterlegt; unerlaubte Übergänge werfen eine Ausnahme

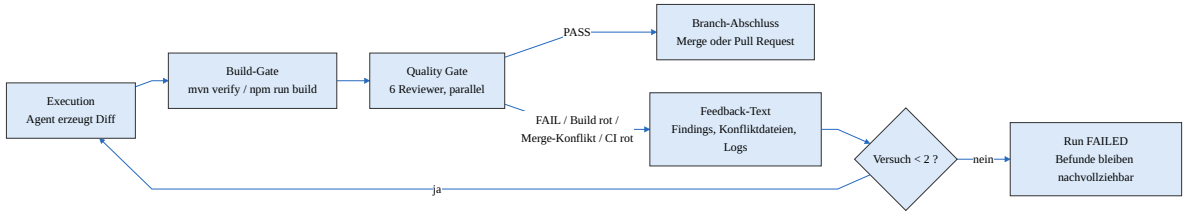

Abbildung 19.7: Die Korrekturschleife als Regelkreis — Befunde werden zu Eingaben des nächsten Laufs

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

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

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

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

Der Regelkreis: Korrekturschleife statt Retry

Kapitel 7 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,[3] hinter der zehn Adapter stehen — ein deterministischer `mock`-Adapter (Default; macht das System ohne Vendor demonstrierbar und testbar), sechs CLI-Adapter (Claude Code, Codex, Gemini, Aider, Kimi, lokales LLM z. B. via Ollama) und drei Cloud-Gateways (AWS Bedrock, Google Vertex AI, Azure OpenAI). Die Adapterwahl ist eine Konfigurationsfrage je Run, aufgelöst über eine Hierarchie (Run-Override > Projekt-Default > User-Setting > globales Setting > Konfigurationsdatei); ein Projekt kann die erlaubten Adapter einschränken, und Policy-as-Code kann diese Wahl mandantenweit übersteuern.

Drei Entwurfsentscheidungen im Port selbst:

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

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

Abo-Modus statt Token-Abrechnung

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

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

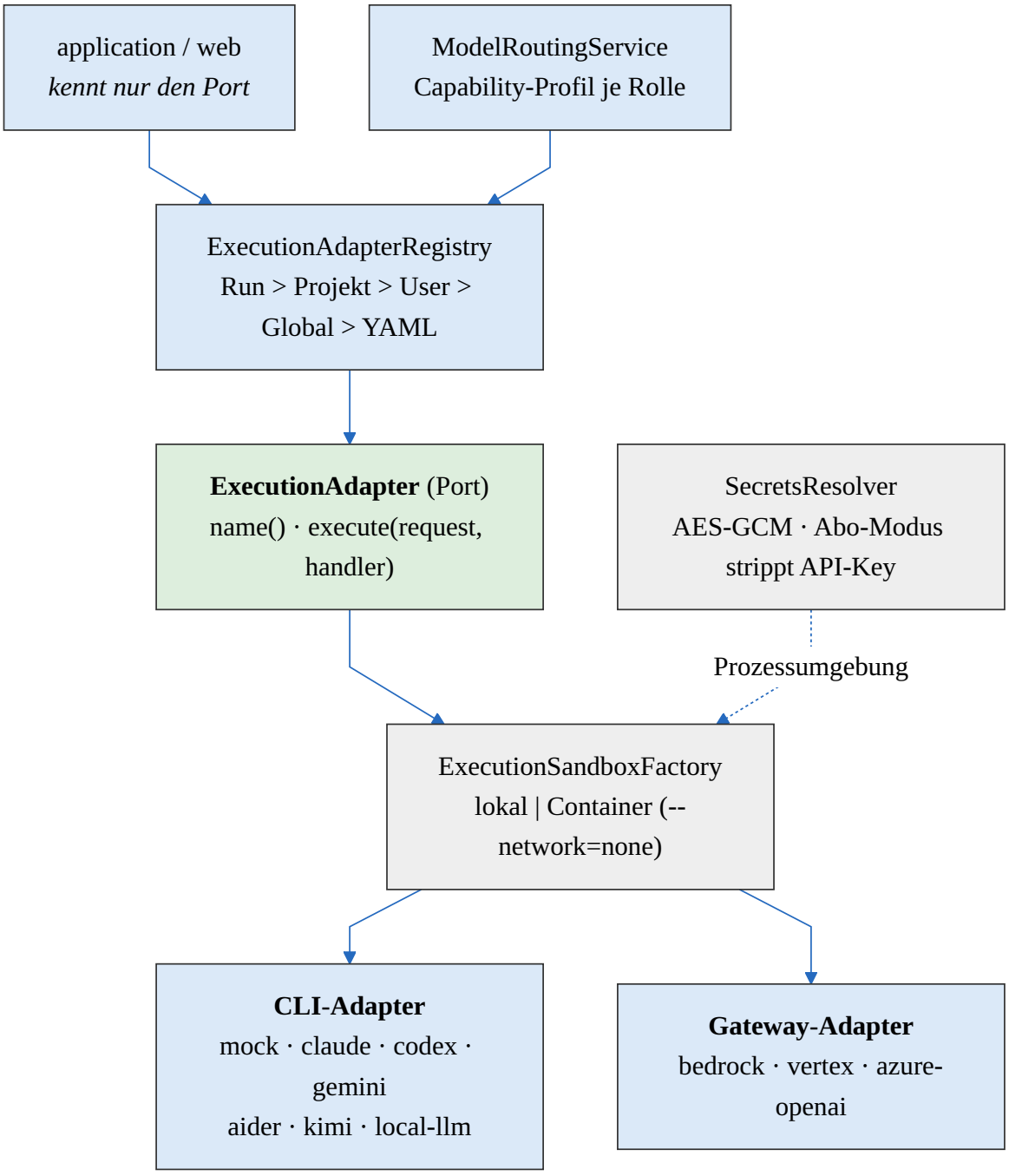

Abbildung 19.8: Vendor-Neutralität durch einen Port: Application- und Web-Schicht kennen ausschließlich den ExecutionAdapter (ArchUnit-erzwungen)

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

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

Capability-Routing statt Modellnamen

Die Modellwahl-Strategie aus Kapitel 5 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 irgendwo ein Modellname im Code steht — wichtig in einem Markt, in dem Modellnamen eine Halbwertszeit von Monaten haben. Eine je Projekt gesetzte Modell-Policy wird beim Anlegen eines Runs erzwungen und attestiert; im Nachhinein ist belegbar, welches Modell wirklich gearbeitet hat.

Sandbox

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

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

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

19.5 Guardrails in der Praxis

Die Guardrails-Pipeline aus Kapitel 12 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.

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

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

Die Gate-Policy

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

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

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

Das Confidence Scoring aus Kapitel 12 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.

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

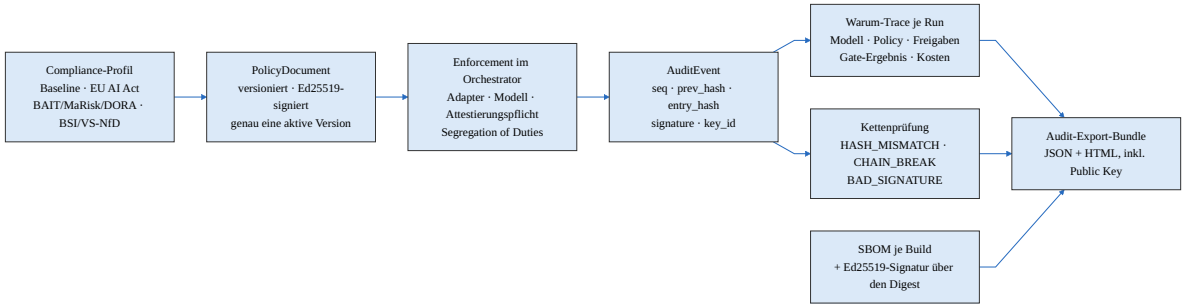

Abbildung 19.9: Von der Policy bis zum Auditbericht: jede Durchsetzung erzeugt ein signiertes Kettenglied

Drei Betriebsmodi

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

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

19.6 Governance und Nachweisführung

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

Mandanten und Rollen

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

Das fünfstufige Rollenmodell (Viewer < Developer < Maintainer < Owner < Admin) entspricht dem Least-Privilege-Modell aus Kapitel 14 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.[6] Je Mandant existiert **genau eine aktive Version** — erzwungen über einen partiellen Unique-Index in der Datenbank plus Service-Logik. Ohne diese Invariante wäre die Frage „welche Regel galt zu diesem Zeitpunkt?“ nicht beantwortbar. Durchgesetzt wird im Orchestrator, an den zwei in 19.3 beschriebenen Zeitpunkten.

Compliance-Profile

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

Profil	Gate	Pflichtfreigaben	Attestierung	Regulatorischer Bezug
Baseline	advisory	—	nein	kein reguliertes Umfeld
EU AI Act	blocking	Execution	ja	VO (EU) 2024/1689, u. a. Art. 12 (Aufzeichnungspflichten), Art. 14 (menschliche Aufsicht) [3]
BAIT / MaRisk / DORA	blocking	Execution, Validation	ja	BaFin-Anforderungen an die IT; DORA (EU) 2022/2554 — IKT-Risiko, Nachweisführung [2]

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

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

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

Die signierte Audit-Hashkette

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

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

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

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

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

Warum-Trace und Audit-Export

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

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

Lieferkette und Wissensbestand

Der Supply-Chain-Nachweis aus Kapitel 14 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 Quelle, mehrere Projektionen** statt einer gepflegten Kopie je Werkzeug; welche Guardrails-Version gewirkt hat, wird attestiert.

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

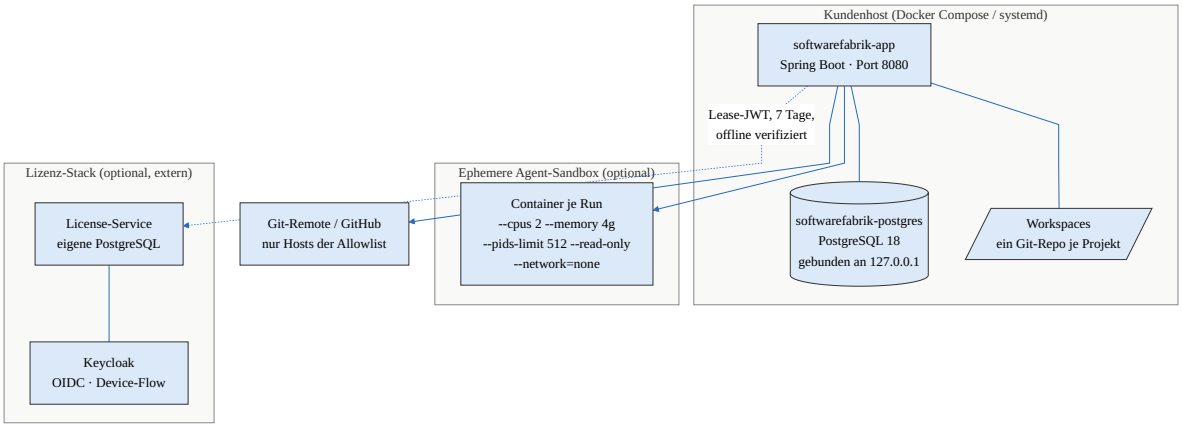

Abbildung 19.10: Deployment-Sicht: ein Anwendungscontainer, eine Datenbank, optional getrennter Lizenz-Stack. Air-Gap-fähig

Gedächtnis ohne Vektorspeicher

Das Fünf-Schichten-Memory-Modell aus Kapitel 8 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 und optional ephemere Agent-Container je Run. Der Lizenz-Stack (OIDC-Provider plus Lizenzdienst) ist getrennt und optional; er kann außerhalb der Unternehmensgrenze stehen oder ganz entfallen.

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

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

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

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

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

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

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

19.8 Was die Praxis am Konzept korrigiert hat

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

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

Das Konzept (Kapitel 1, 3, 10 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 Parallelität deshalb von der Erzeugung auf die Bewertung. Aus demselben Grund wurde die Workspace-Isolation über Git-Worktrees (ADR-2) zur **Branch-Isolation auf einem projektpersistenten Workspace**: Iteration über

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

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

Das Konzept beschreibt eine Retry-Strategie (Kapitel 7.5, 9 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 Lizenz-Stack, lokales Modell, netzlose Sandbox — verschiebt die Komplexität dorthin, wo sie hingehört: in die Steuerung des Nichtdeterminismus, nicht in die Betriebsinfrastruktur.

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

als Orchestrierungsmechanismus) gilt mit einer Präzisierung: Git ist Zustandsträger — Branch, Commit, Checkpoint, Merge, PR —, aber nicht die alleinige Wahrheit. Der autoritative Zustand liegt in der Datenbank, weil ein Auditor Fragen stellt, die `git log` nicht beantwortet: welche Policy, welches Modell, wessen Freigabe.

19.9 Grenzen und offene Punkte

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

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

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

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

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

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 0.30.0

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

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

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

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

Das Zielbild

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

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

Die tragenden Prinzipien

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

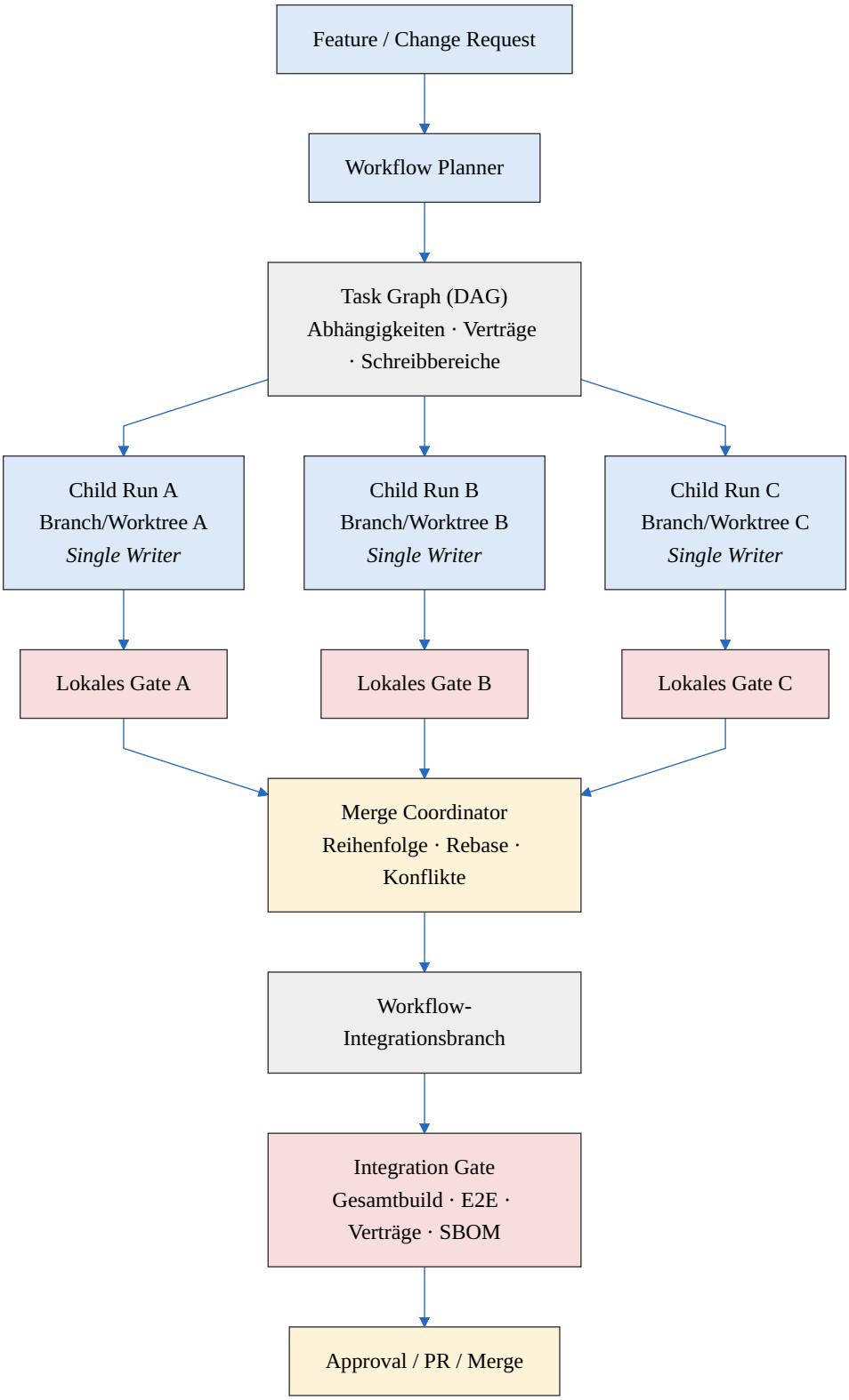

Abbildung 19.11: Architektur der parallelen Agenten-Workflows: isolierte Child Runs unter einem Parent Workflow, zusammengeführt über Merge Coordinator und Integration Gate. Bis Release 0.30.0 umgesetzt, hinter deaktiviertem Feature-Flag

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

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

Die Ausbaustufen

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

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

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

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

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

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

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

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

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

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

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

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

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

Stufe 4a (Release 0.24.0) macht Planänderungen nachweisbar. Bisher sagte die Planversion, *welcher* Plan gilt — nicht, was sich geändert hat und warum; genau das wird aber gefragt, meist Wochen später. Jede Änderung liegt nun unveränderlich als Revision vor, mit Grund, Pflichtbeschreibung, Auslöser und betroffenen Tasks, attestiert über die Audit-Hashkette. Bemerkenswert ist, wie hier das Rule-Loop-Prinzip (Regelkreis statt blindem Retry, Kapitel 2 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. Der Ehrlichkeitsanker ist die **Aufwandsabdeckung**: Neben der Summe steht immer, aus wie vielen Entscheidungen sie stammt — „45 min (aus 3/12)“ macht sichtbar, dass neun Entscheidungen keine Angabe tragen; ohne diese Quote würde die Summe als Gesamtaufwand gelesen, also genau die Verwechslung, gegen die die Lücke überhaupt formuliert wurde. Bewusst **nicht** gebaut: Interaktions-Telemetrie und Leerlauf-Erkennung — das ergäbe eine Zahl, aber um den Preis, Menschen zu vermessen, und sie wäre trotzdem geraten. Die Arbeitszeit selbst bleibt damit ausgewiesene Messlücke.

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

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] *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).

[3] *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).