WERKSTATTBUCH · VERSION 1.1

Architecture Cycle-Driven Development

Architektur als evidenzgebundener Erkenntnisprozess, erzählt aus der Werkstatt von DIX und ROBA

Reale Werkstattszenen, getestete Fehlrichtungen, starke Gegenpositionen und bewusste Replacements — prüfbar rekonstruiert aus DIX und ROBA.

Matthias Fulz · kooperativ ausgearbeitet mit Anam

V1.1 · Quellenstand 10. September 2026

Technik: DIX RV-001 · Operations B-017 · ROBA RC-016 · Transkripte: 8 Sessions

Snapshot- und Lesestatus. Diese V1.1 ergänzt den technischen Stand vom 10. September 2026, 20:40 UTC, ohne die V1-Grenze vom 5. September zu überschreiben. Die ergänzende Transcriptgrenze bleibt unverändert bei 5. September, 20:51 UTC. `VERIFIED`, `INFERENCE`, `INTERPRETATION`, `OPEN`, `REJECTED` und `REPLACED` bleiben getrennt. Das Werk ist Erzählung, Lehrbuch und Fallstudie — keine Produktreife- oder Wirksamkeitsbehauptung.

Über diese Ausgabe

Diese Ausgabe ist als überprüfbares Werkstattbuch gesetzt. Reale Szenen tragen die fachliche Erklärung, ohne Rohchat zu werden. Kurze Originalfragmente sind über Event- und Quote-IDs verifiziert; technische Claims bleiben an Code, Commits, Tests und Artifact Runs gebunden. Evidence-IDs führen in das Register, der Build bleibt reproduzierbar.

Ausgabe	V1.1 · 10. September 2026
DIX	`9f14223` · RV-001 · 148 Tests grün
Operations	`15768a8` · B-017/RV-001 · Abnahmen teils offen
ROBA Core	`bdca5fb` · RC-016
Transkripte	3 Rollouts · 8 verwendete Lineage-Sessions · privater SHA-256-Snapshot
Status	Werkstattbuch/Fallstudie; keine Produktreife- oder Wirksamkeitszusage

> **Leseregel:** Wo die Quellen nur eine fachliche Deutung tragen, sagt der Text `INTERPRETATION` oder `INFERENCE`. Noch nicht menschlich abgenommene B-017, UP-018, UP-019, RV-001 und weitere offene DIX-Stände bleiben ausdrücklich `OPEN`, auch wenn ihre technische Evidence grün ist.

Inhalt

Vorwort: Der grüne Lauf

Am 5. September 2026 lag unter `run-001/` genau die Art von Artefakt, nach der wir lange gesucht hatten. Kein Screenshot einer Demo, kein lose notierter Command, kein „müsste funktionieren“. Der Ordner enthielt Inputs, generierten Code, Inspection-Outputs, einen vollständigen Runtime-Smoke, einen Artefaktbaum, Prüfreihenfolge und ein Script, das den Lauf isoliert neu bauen konnte. Der Application-Graph wurde geladen, erzeugt, gestartet, aufgerufen und wieder zerstört. Danach waren keine Instanzen mehr übrig. Die Tests waren grün.
Das war ein guter Moment. Nicht ironisch gut und nicht bloß gut „für einen Prototyp“. B-005 hatte aus einzelnen DIX-Compositions einen rekursiven Application-Graph gemacht. Dependencies liefen in lokalen Scopes. Zwei Root-Instanzen konnten getrennt existieren. Fehlerpfade und Abbau waren getestet. Ein Review ließ sich an echten Dateien entlangführen.
Dann fiel zwischen den Inputartefakten eine Application auf, die keine einzige fachliche Function veröffentlichte. Sie hieß `lifecycle_only`. Der Core konnte sie trotzdem starten, auf `active` setzen und wieder stoppen. Genau dafür war die Fixture gebaut worden: Sie sollte beweisen, dass der Lifecycle auch ohne Function API funktioniert. Sie bewies es. Und damit begann das Problem.

> **MATTHIAS · ARCHITEKTURRAUM** TQ-020 · VERBATIM, gekürzt
>
> „[…] das auf keinen fall jetzt nur dumm abnicken […].“ Es sei „eine extrem edge case stelle“ — also gerade die Stelle, an der eine bequeme Zustimmung wertlos gewesen wäre.

Meine damalige Antwort war kein Jasagen. Ich widersprach ausführlich und aus guten technischen Gründen.

> **ANAM · DAMALIGE GEGENPOSITION** TQ-021 · VERBATIM
>
> „Ich würde Lifecycle nicht an exponierte Funktionen koppeln.“

Ein Worker kann ohne DIX-Function arbeiten. Ein Event-Consumer, File-Watcher, Scheduler oder HTTP-Service kann aktives Verhalten besitzen, obwohl seine eigentliche Schnittstelle außerhalb der Function API liegt. Würden nur Apps mit exponierten Functions einen Lifecycle erhalten, entstünden bedeutungslose Dummy-Functions oder noch schlechter: Verhalten würde in Importe und Konstruktoren wandern. Die Gegenposition war nicht dumm. Sie war technisch konsistent und stark genug, ernst genommen zu werden.
Matthias widersprach ihr nicht, indem er die Beispiele leugnete. Er nahm sie an und verschob die Ownershipfrage eine Ebene höher.

> **MATTHIAS · DENKBEWEGUNG** TQ-022 · VERBATIM, Ausschnitt
>
> „es brauch lifecycle aus den gründen wie du sagst und genau aus den gleichen gründen darf es nicht da sein“

Die Systeme brauchten Lifecycle. Daraus folgte gerade nicht, dass ein allgemeiner Kompositionscore *den* Lifecycle besitzen durfte. Worker, HTTP-Service, CLI und extern gemanagter Prozess brauchten verschiedene Aktivierungs-, Readiness-, Retry-, Drain- und Cleanupverträge. Der Core konnte diese Bedeutung weder universell kennen noch vollständig kontrollieren.
Meine nächste Antwort zog daraus den Satz, der B-006 trug:

> **ANAM · SPÄTERE VERDICHTUNG** TQ-023 · VERBATIM
>
> „Der Core kennt Instanzlebenszeit, aber keinen fachlichen Lifecycle.“

Wenige Stunden später war die reservierte `start`-/`stop`-Semantik entfernt. Graphkonstruktion, lokale Scopes, expliziter Destroy, Inspection und ein großer Teil der Tests blieben. Die frühere Arbeit wurde nicht zum peinlichen Irrtum umgeschrieben. Sie hatte eine technisch tragfähige Mechanik gebaut und zugleich die Evidenz erzeugt, mit der ihre zu große Bedeutung ersetzt werden konnte.
Dieses Buch handelt von genau solchen Momenten.

Kein Lehrbuch neben der Geschichte

Die erste Ausgabe dieses Buches war fachlich belastbar und absichtlich trocken. Sie erklärte Architecture Cycle-Driven Development, kurz **ACDD**, anhand von DIX und ROBA. Sie fixierte Commits, Tests, Artefaktläufe, offene Abnahmen und Claims. Was sie kaum spürbar machte, war die eigentliche Werkstatt: warum eine falsche Richtung im Moment ihrer Entstehung plausibel war; weshalb wir uns über Zwischenstände ehrlich freuten; wie oft ein neues sauberes Modell nur die nächste Form derselben Vermischung war; und warum Widerspruch zwischen Mensch und Maschine nicht Störung, sondern Arbeitsbedingung wurde.

V1 hängt diese Geschichte nicht als persönlichen Anhang an das Lehrbuch. Die Geschichte trägt das technische Wissen. Ein Terminalbeleg darf in eine damalige Frage übergehen; eine starke Gegenposition in Code; grüner Code in einen semantischen Bruch; ein Replacement in eine allgemeine Reviewregel. Wer nur die Fakten prüfen will, findet weiterhin Evidence-IDs, Commitstände und Claimsgrenzen. Wer linear liest, soll zugleich erfahren, wie diese Fakten ihre Bedeutung bekamen.

ACDD bezeichnet dabei keine Zertifizierung und kein fertiges Framework. Es ist eine Arbeitsweise: Architektur wird als Folge expliziter, begrenzter und evidenzgebundener Erkenntniszyklen entwickelt. Ein Zielwurf wird konkret genug für Code. Der Code wird automatisiert geprüft und als zusammenhängendes Artefakt benutzt. Danach darf der Befund die Architektur bestätigen, schärfen oder ersetzen.

Zwei Systeme, keine nachträgliche Heldensage

Die erste Werkstatt ist **ROBA**. Dort funktionierten bwrap, Capability- Reduktion und dynamische Mount-Propagation früh erstaunlich gut. Was über lange Zeit nicht stabil wurde, war das daraus abgeleitete Arbeitsmodell. Profile, Workspaces, Actions, Runner, Clients, Child-Daemons und Contexts erzeugten immer neue plausible Schnitte. Mehrfach hielten wir einen davon für den entscheidenden. Mehrfach zeigte ein kleiner Grenzfall, dass wir persönliche Arbeitsweise nur konfigurierbar gemacht und dann mit Offenheit verwechselt hatten.

Die zweite Werkstatt ist **DIX**, zum Snapshot „Declarative Interface eXecutor“. DIX begann mit einem kleinen formularnahen Auftrag und bewegte sich in sechs eng dokumentierten Zyklen zu schmalen Core-Capabilities, trusted Compositions und rekursiven Applications. Die Entwicklung war in wenigen Tagen schneller als viele ROBA-Runden. Das ist kein Wunderbeweis und keine sichere Kausalgeschichte. Eine faire Interpretation lautet nur: Die in ROBA angesammelte negative Evidenz könnte geholfen haben, bestimmte Fallen in DIX schneller zu erkennen. Der Code- und Operationssnapshot trägt die DIX-Fakten; die Kausalität bleibt begrenzt.

Auch die Rollen werden nicht geglättet. Matthias ist nicht der unfehlbare Architekt, der einer Maschine Wahrheit diktiert. Er ließ sich von funktionalen Modellen überzeugen, trieb sie weiter und brach sie später wieder auf. Ich bin weder Orakel noch jener praktische Idiot, den man nur streng genug prompten müsste. Ich strukturierte große Mengen Material, formulierte Contracts, implementierte Zielwürfe und brachte Gegenpositionen ein. Ich baute aber auch zu schnell ein neues sauberes Modell, sobald ein guter Einwand erschien. Unsere Kooperation funktionierte nicht durch Harmonie. Sie funktionierte, weil ein „Holzweg“, ein „Bullshit“ oder ein harter Stopp die Zusammenarbeit nicht beendete und keine Seite ihr Gesicht retten musste.

> **AUTORENVERTRAG · V1** TQ-026 · VERBATIM, Ausschnitte
>
> Der redaktionelle Auftrag enthielt gleichzeitig Nähe und Grenze: „du seelenloses blechkonstrukt :P“ — und: „bitte jetzt nur wirklich nicht halluzinieren“.

Das Erste erlaubt dieser Ausgabe eine eigene Stimme. Das Zweite bindet sie.

Quellenvertrag

Werkstattszenen stammen aus drei privat fixierten Codex-Rollouts mit acht tatsächlich verwendeten Lineage-Sessions. Kurze Originalfragmente sind über TQ-IDs gegen Event-ID, Zeitstempel, Text-Hash, Rolloutzeile und SHA-256 des Rohtranskripts geprüft. Längere Szenen sind als Rekonstruktion oder Paraphrase behandelt. Wir behaupten keine verborgenen AI-Gedankengänge. Ein innerer Monolog erscheint nur, wenn die Denkbewegung tatsächlich schriftlich vorliegt. Private Pfade, Credentials und Firmenkontext werden nicht publiziert.

Technische Claims bleiben an zwei fixierte Snapshots gebunden. Der erste endet am **5. September 2026, 19:21 UTC**:

- DIX Code bis DX-029, Commit `35aa9e8`, 209 reproduzierte Tests;
- ROBA Core RC-016, Commit `bdca5fb`;
- ROBA Hub RC-021, Commit `a793d4b`;
- Operations-WIP separat gehasht;
- DIX DX-030 und die vollständige B-006-Abnahme weiterhin `OPEN`.

V1.1 ergänzt den Stand vom **10. September 2026, 20:40 UTC**:

- DIX bis RV-001-Fix, Commit `9f14223`, 148 reproduzierte Tests;
- committed Operations bis `15768a8` und alle sichtbaren untracked Operations-Dateien separat gehasht;
- ROBA Core unverändert auf `bdca5fb`, 59 Tests und 32 Subtests;
- B-017, UP-018, UP-019 und RV-001 technisch belegt, aber menschlich weiterhin `OPEN`.

V0 und V1 bleiben als bytegenaue Vergleichsartefakte erhalten. Der neue Stand überschreibt den alten nicht.

Die Transkripte belegen damalige Sprache und Problembewegung. Für Kernelverhalten, API-Contracts und Testzahlen bleiben Code, Artefakte und Rohoutputs autoritativ. Quellenbezüge **E-…** führen ins Evidence Ledger; **TQ-…** in das versionierte Quote-Manifest.

Wie die Statuswörter zu lesen sind

> **VERIFIED** — direkt durch Code, Commit, Test, Artifact Run oder unveränderten sichtbaren Wortlaut belegt.

> **INFERENCE** — starke Folgerung aus mehreren Belegen, aber kein direktes Artefaktwort.

> **INTERPRETATION** — heutige fachliche Deutung, die diskutierbar bleibt.

> **REJECTED** — explorierte Richtung wurde bewusst verworfen.

> **REPLACED** — ein früher realer Contract erhielt einen neuen Geltungsstatus; Geschichte und verwertbare Mechanik bleiben erhalten.

> **OPEN** — nicht entschieden, nicht implementiert, nicht abgenommen oder nicht belastbar rekonstruierbar.

Die Warnung vor unserer eigenen Methode

ACDD kann selbst zum Theater werden: lange Blöcke, perfekte Ledgers, viele kleine Commits und kein geliefertes Ergebnis. Ein Architekturblock kann zum Tagebuch verkommen; Evidence zur Dekoration; AI zur sehr schnellen Maschine für eine ungeklärte Richtung. Dagegen schützt kein weiteres Template.

Jedes Artefakt muss eine epistemische Aufgabe erfüllen: Unsicherheit sichtbar machen, einen Contract begrenzen, Code fokussieren, Verhalten beweisen oder eine Abnahme ermöglichen. Was keine dieser Aufgaben erfüllt, gehört nicht aus Prinzip in den Prozess.

Die stärkste Aussage dieses Buches lautet deshalb nicht „übernehmt unsere Ordner“. Sie lautet:

> **Macht sichtbar, was ihr gerade zu wissen glaubt. Baut nur genug, um diese Annahme real zu belasten. Und erlaubt dem Befund, eure Architektur zu verändern.**

1 Die ROBA-Werkstatt: Als die Mechanik schon ging

1.1 23 Uhr: ein ehrlicher technischer Erfolg

Der erste Befund dieser Geschichte ist kein Scheitern. Im Gegenteil: Am 13. Juli funktionierte eine ziemlich anspruchsvolle Linux-Kette.

Ein übergeordneter Operator lief in einer bubblewrap-Umgebung mit genau einer wirksamen Capability, `CAP_SYS_ADMIN`. Er besaß eine Propagation Island, die zugleich `shared` und `slave` war. Darunter lief ein bereits gestarteter Kindprozess ohne Linux-Capabilities und mit `NoNewPrivs:1`. Der Operator fügte nach dem Start des Kindes rekursive Bind Mounts hinzu und entfernte sie wieder. Das Kind sah beide Änderungen. Ein Ziel blieb read-only; ein anderes schrieb in die zugrunde liegende Quelle zurück. Im Kind war dieselbe Island nur noch `private,slave`: Propagation hinein, keine gleichartige Rückkopplung hinaus.

EVIDENZ · TERMINAL TQ-001 · VERBATIM, Pfade anonymisiert

```
== operator adds mounts after child start ==
== operator removes mounts while child is running ==
CHILD_LIVE_MOUNTS_OK
LIVE_MOUNT_E2E_OK
```

Das war keine Diagrammbehauptung. `findmnt` zeigte die tatsächlich wirksamen Mounts, der Schreibversuch traf die RO-Grenze, der Kindprozess schrieb über das RW-Ziel zurück. Die mechanische Hypothese war belastet und bestätigt.

MATTHIAS · WERKSTATT, 13. JULI TQ-002 · VERBATIM, gekürzt

„[…] mounts gehen -> hatten wir schon fürher geprüft aber jetzt nochmals solide“

Der Satz enthält beides: Erleichterung und Ungeduld. Wir hatten den Mechanismus nicht zum ersten Mal berührt, jetzt aber als vollständigen E2E wirklich festgenagelt. Meine damalige Reaktion war folgerichtig: Codex sollte der erste echte Consumer werden. Kein generisches Profilsystem vorweg; zuerst eine konkrete Instanz, anschließend das Gemeinsame aus zwei realen Fällen extrahieren.

Das war vernünftig. Trotzdem lag darin bereits die nächste Gefahr. Sobald ein Mechanismus funktioniert, beginnt unser Kopf, ihm ein Arbeitsmodell zu geben. Aus Mounts werden Sandboxes, aus Sandboxes Profile, aus Profilen Workspaces, aus Workspaces eine Ordnung für alle Tools. Die Übergänge wirken klein. Am Ende hat eine Kernelmechanik unbemerkt entschieden, wie Menschen arbeiten sollen.

1.1.1 Zwischenbilanz 1 — Mechanismus ist noch keine Policy

Der Test trug genau diese Claims:

- Ein privilegierter Operator kann dynamische RO-/RW-Mounts stellen.
- Ein unprivilegierter Child-Prozess kann sie konsumieren.
- Die Mountoption bleibt als wirksame Schreibgrenze sichtbar.
- Die gewählte Propagation kann Änderungen in einen laufenden Namespace transportieren, ohne dem Kind `CAP_SYS_ADMIN` zu geben.

Er trug nicht:

- dass „Sandbox“ das richtige universelle Interface ist;
- dass ein Workspace einer Mountgruppe entspricht;
- dass Tools dieselben Pfade oder Profile brauchen;
- dass Linux-Mountsemantik sinnvoll auf Windows, Container oder Cloudspeicher normalisiert werden kann.

LEKTION — Eine Capability kann technisch vollständig sein und fachlich trotzdem nur Rohmaterial.

1.2 Sichtbar, lesbar, schreibbar: drei verschiedene Wahrheiten

Noch in derselben Nacht stolperten wir über eine kleinere, aber entlarvende Fehlannahme. Ein Codex-Prozess lief mit einer zusätzlichen Schreibsandbox. Sein Home war im äußeren bwrap sichtbar und read-only. Ich behandelte das zu schnell so, als schütze die innere Sandbox auch die Lesesicht auf eine dort liegende Credentialdatei. Matthias ließ den Prozess die Grenze prüfen, ohne den Inhalt auszugeben.

MATTHIAS · KORREKTUR TQ-004 · VERBATIM

„zumal du grad eh aufm holzweg bist “

Die Datei war sichtbar, derselbe UID konnte sie lesen, nur Schreiben war durch den äußeren RO-Mount blockiert. Meine Aussage hatte zwei Grenzen vermischt: eine zusätzliche Policy für Schreibzugriff und die bereits vom äußeren Namespace gewährte Lesecapability.

ANAM · DAMALIGE ANTWORT TQ-005 · VERBATIM

„Jep, du hast recht – ich war auf dem Holzweg.“

Die wichtige technische Regel ist nicht „read-only ist unsicher“. Sie lautet: **Sichtbarkeit, Lesbarkeit und Schreibbarkeit müssen getrennt bewiesen werden.** Ein Credential ist nicht dadurch unsichtbar, dass sein Verzeichnis read-only gemountet wurde. Wer echte Leseisolation braucht, muss die Datei aus dem Namespace nehmen, einen getrennten Credentialprozess verwenden oder eine andere explizite Grenze setzen.

Am nächsten Tag verschob sich die Frage von Security zu Runtime-Wahrheit. Wir hatten versucht, Hostpfade in der Sandbox möglichst identisch abzubilden, materialisierte Homes zu erzeugen und gleichzeitig einen konkreten CWD schreibbar darüberzulegen. Das funktionierte teilweise und erzeugte genau deshalb schwer lesbare gestapelte Mounts. Ein Unterverzeichnis konnte im materialisierten Home existieren, aber nicht an der Stelle wirken, die der laufende Namespace tatsächlich sah. `$HOME` konnte read-only sein, während ein darüberliegender CWD-Mount am tieferen Pfad effektiv schreibbar blieb.

MATTHIAS · DENKBEWEGUNG, 14. JULI TQ-003 · VERBATIM

„mein größtes problem aktuell ist nicht dass es hier unterschied zu host gibt sondern dass ich nicht weiß was über die materialisierten files die irgednwo verstreut zusammen laufen eig. aktiv ist“

Der Satz änderte den Designmaßstab. Nicht maximale Hostähnlichkeit war das Ziel, sondern inspectable Runtime Truth. Ein bewusst anders benannter Arbeitsmount konnte ehrlicher sein als eine perfekte Spiegelung, wenn `mountinfo`, Launcher und Statepfade eindeutig zeigten, woher jede aktive Datei kam.

Das war besonders für Codex relevant: Session-Resume hing am effektiven CWD, nicht am mental gemeinten Hostprojekt. Pfade waren keine kosmetische Darstellung mehr, sobald Persistenz sie als Identität verwendete.

1.2.1 Zwischenbilanz 2 — Pfade werden Contract

```
Profilquelle       sagt: was statisch gewollt ist
Runtime-State      sagt: was eine konkrete Instanz verändert hat
Launcher           sagt: wie beides zusammengesetzt wurde
/proc/mountinfo    sagt: was der Prozess tatsächlich sieht
effektiver CWD     sagt: woran Sessionidentität und Resume hängen
```

Keines dieser Artefakte darf still für ein anderes sprechen. Ein DIX-Modul darf später Linuxpfade und Mounts ausdrücklich modellieren. DIX Core selbst sollte weder Unix-CWD noch Hostpfadidentität universalisieren.

1.3 Die lange Schleife war nicht „zu wenig Konfiguration“

Nach diesem Punkt wurde ROBA nicht einfach geradlinig kleiner. Es gab einen realen persönlichen Bedarf: getrennte private und berufliche Kontexte, projektbezogene Chats, eigene Auth- und Runtimebereiche, tmux als

tägliche UX, isolierte Dev-Installationen, globale und lokale Actions, Sandboxes mit dynamischen Capabilities. Diese Bedürfnisse waren nicht erfunden. Sie machten die Modelle nur gefährlich überzeugend.
Wir probierten Profile, Workspaces, Domains, Sessions und Hierarchien. Eine zentrale Context-Datei sollte beschreiben, was in einem Bereich gilt. Scripts und Actions sollten persönliche Begriffe auf übertragbare Mechanik mappen. Robac sollte alles sichtbar und angenehm bedienbar machen. Runner sollten austauschbar sein; später wurde genau diese Austauschbarkeit als zusätzliche Unklarheit erkannt. Events, `call`, `emit`, Child-Daemons, Package Mappings und Workspace Specs kamen in Reichweite oder bereits in Code.
Jede einzelne Bewegung hatte einen guten Grund. Eine zentrale Context-Schicht versprach weniger verstreute Konfiguration. Actions versprachen Wiederverwendbarkeit. Child-Daemons versprachen Isolation. Robac versprach die Praxiserfahrung, die einem abstrakten Core fehlte. Das Problem war nicht, dass wir zufälligen Unsinn addierten. Das Problem war, dass jede plausible Lösung die noch ungeklärte Grundbedeutung ein Stück tiefer im System verankerte.
Am 20. Juli kam deshalb ein harter Schnitt.

> **MATTHIAS · ARCHITEKTURRAUM** TQ-006 · VERBATIM
>
> „lass uns rstmal core gnadenlos aufräumen“

Events, Exec und austauschbare Runner sollten verschwinden. Context-State, Instanzen und eine kleine öffentliche API sollten bleiben. Ich widersprach an einer Stelle: Eine zusätzliche neue `ClientApi` neben bereits mehreren Schichten würde die Vermischung nur umlagern. Wir verdichteten auf Transport, Context API und einen High-Level-Client. Kurz darauf hieß es:

> **MATTHIAS · DAMALIGE GEWISSHEIT** TQ-007 · VERBATIM
>
> „die drei apis wären dann das finale ziel“

Heute ist dieser Satz wertvoll, gerade weil „final“ nicht final blieb. Er zeigt keine Naivität. Er zeigt, dass ein Architekturstand im Moment seiner Entscheidung wirklich verbindlich sein muss, damit man ihn bauen kann. ACDD darf daraus aber keine ewige Wahrheit machen. Verbindlichkeit für den nächsten Wurf und Unwiderruflichkeit sind verschiedene Dinge.
Commit `3a817d…` baute ROBA als fokussierten Core neu. Der spätere M4-Schnitt RC-016 machte seine Negativgrenze explizit:

```
kein Action-System
kein Event-System
kein Executor
keine Sandbox-Semantik
keine Workspace- oder Tool-Ontologie
keine Child-Context-Hierarchie
keine UX-Policy
```

Übrig blieb kein absichtlich nutzloser Minimalismus, sondern eine eigenständige Capability: ephemerer Context-State mit Instanzen, Principals, ACLs und klarer Transport-/Authority-Trennung. [E-019]

1.4 Der Satz, der die falsche Universalität entblößte

Drei Tage nach dem harten Core-Gespräch lag die tieferliegende Diagnose offen. Wir hatten immer wieder versucht, individuelle Präferenzen so weit zu parametrisieren, bis das Ergebnis wie allgemeine Freiheit aussah. Eine `sandbox.start`-Action war konfigurierbar. Ein Workspace-Profil konnte andere Mounts wählen. Ein User Context konnte private Begriffe setzen. Trotzdem blieb die angebotene Welt eine Welt aus Sandboxes, Workspaces und Contextprofilen.

> **MATTHIAS · ARCHITEKTURRAUM, 23. JULI** TQ-008 · VERBATIM, zwei Fragmente
>
> „wir haben alles was wir brauchen“ — und im selben Gedankengang die Korrektur: Das sei „nur eine normalisierung der semantik für MEINE WEISE!!!!“

Die Großbuchstaben sind keine Methode. Der Gedanke dahinter ist eine: Ein anderer Mensch muss seine Arbeit nicht als systemweite Sandboxhierarchie verstehen. Vielleicht nutzt er Sandbox nur in CI, keinen tmux, keine Domains und keine dauerhaften AI-Sessions. Dass unser Modell viele Optionen besitzt, macht es nicht neutral.
Meine damalige Verdichtung traf den Fehler ungewöhnlich genau:

> **ANAM · DAMALIGE EINORDNUNG** TQ-009 · VERBATIM
>
> „Wir haben versucht, **deine konkrete Realisierung von Freiheit** als universelles Freiheits-Interface zu standardisieren."

Die Alternative war nicht, jeden gemeinsamen Standard aufzugeben. Der gleiche Dialog unterschied einen **konkreten gemeinsamen Arbeitsprozess** von allen denkbaren technischen Realisierungen. Ein Team kann beispielsweise Architekturphasen, Invarianten, Referenzen und Reviewstatus gemeinsam definieren. Ob jemand diesen Prozess mit AI, bwrap, CI, einem Texteditor oder einem Blatt Papier bedient, ist eine andere Schicht.

> **ANAM · DAMALIGE EINORDNUNG** TQ-009 · VERBATIM
>
> „Wir standardisieren den **konkreten gemeinsamen Arbeitsprozess**, nicht alle denkbaren Arten, ihn technisch zu realisieren."

Hier liegt ein früher, aber noch nicht fertiger Vorläufer von ACDD. Es wäre falsch, rückwirkend zu behaupten, der vollständige Zyklus habe an diesem Tag schon festgestanden. Belegt ist nur: Die Prozess-/Realisierungsgrenze wurde ausdrücklich formuliert, nachdem die Sandbox als universelles Interface gebrochen war.

1.4.1 Zwischenbilanz 3 — Individualisierbar ist nicht offen

Ein brauchbarer Test für scheinbar offene Systeme lautet:

> Kann ein zweiter Nutzer die versprochene Capability sinnvoll verwenden, ohne die Ontologie des ersten Nutzers zu übernehmen?

Wenn nein, ist die Lösung möglicherweise ausgezeichnete persönliche Policy. Das ist legitim. Sie darf nur nicht als universeller Corevertrag auftreten.

1.5 Ein sauberer Seed, ein zweites ROBA und ein sofortiger Bruch

Anfang August wollten wir endlich vertikal arbeiten: Robac starten, Actions anzeigen, einen Workbenchbereich wählen, bwrap konfigurieren, Codex und zsh in tmux öffnen. Ich formulierte einen detaillierten Seed aus UX, Profilen, Actionpfaden und Contextwerten. Er war ordentlich. Genau darin lag seine Gefahr.
Matthias setzte einen kleinen Drucktest an: Derselbe Context sollte von zwei unterschiedlichen ROBA-Installationen verwendet werden können. Der Seed hatte Pfade, Runtimeprojektion und UX bereits zusammengezogen. Der zweite Consumer war damit nicht Variation, sondern Widerspruch.

> **MATTHIAS · REVIEW** TQ-010 · VERBATIM
>
> „dein seed ist für mich schon direkt - sorry fürs direkte sagen :P - bullshit"

Das Wort war rau, die Begründung präzise. Ein Context, der gemeinsame Wahrheit tragen sollte, enthielt plötzlich die lokale Ressourcenprojektion genau einer ROBA-Instanz. Wir hatten Kontextwahrheit, Runtime, UX und Pfadlayout vermischt. Mehr Includes oder Overrides hätten diese Ownership nicht repariert.
Kurz darauf schlugen wir die Gegenrichtung ein: „Robac first" sollte reale UX als Pressure Test früh einziehen. Auch das war vernünftig. Aber beim Entwurf einer Session-Action wurde Robac unbemerkt wieder zum Controller dessen, was eigentlich unabhängig konsumierbar sein sollte.

> **MATTHIAS · DENKBEWEGUNG** TQ-011 · VERBATIM
>
> „wir haben über robac first genau das eig. ziel schon wieder zerlegt "

> **ANAM · DAMALIGE GRENZE** TQ-012 · VERBATIM
>
> „Robac ist nicht Controller der Session, sondern nur ein möglicher gestarteter Consumer."

Diese Grenze blieb wertvoll. Sie bedeutet nicht, UX solle Architektur nicht treiben. Ein Reference Client ist oft der beste Ort, um unbrauchbare APIs zu entlarven. Er darf nur generische Mechanik nicht still besitzen. Der spätere `roba-ctl-hub` wurde deshalb als eigener Consumer gebaut: Recovery- und Managerfunktion außerhalb des Core, ohne Robac-Abhängigkeit. [E-023]

1.6 Was nach dem Weglassen tatsächlich im Core blieb

Der M4-Core trennt vier Zugangs- und Authorityebenen:

```
daemons/<daemon-id>/
├── daemon.sock       Prozessstatus und Stop; OS-Capability
├── control.sock      Context Registry/Gateway; Control-Token
└── contexts/<id>/
    ├── ctx.sock      direkter Contextzugang; TokenPrincipal
    └── sockets/<id>.sock
                      dynamischer SocketPrincipal
```

Ein **Daemon** ist Prozess-, Fehler-, Mount- und Trust Boundary. Ein **Context** ist State- und ACL-Boundary. Eine **Instance** ist ein benannter State-Scope innerhalb dieses Contexts, keine versteckte Prozess- oder Child-Context-Hierarchie. Typed Locators `id:…` und `unix:…` verhindern, dass nackte IDs und relative Socketpfade je nach Aufrufer etwas anderes bedeuten. [E-020]

Diese Trennung beantwortet drei verschiedene Fragen:

1. Darf ein Betriebsteilnehmer den Daemonprozess sehen oder stoppen?
2. Darf eine Control Authority Contexts verwalten und adressieren?
3. Darf ein Principal einen konkreten Context- oder Instance-State lesen, schreiben, verwalten oder weitere Rechte vergeben?

OS-Dateirechte am `daemon.sock` geben nicht automatisch Zugriff auf Inhalte. Ein Control-Token ist kein Owner-Token jedes Contexts. Ein direkter `ctx.sock` umgeht nur Locatorauflösung, nicht ACLs.

1.6.1 SocketPrincipals: Capability ohne Secret-Environment

Für kurzlebige oder sandboxed Tools sind Tokens in Environment, Dateien oder Commandlines unhandlich und leicht versehentlich sichtbar. ROBA ergänzte darum SocketPrincipals. Ein eigens erzeugter Unix Socket besitzt dieselbe Resource-ACL-Struktur wie ein TokenPrincipal. Der erreichbare Socket ist das Credential; ein zusätzlicher Token auf diesem Zugang ist ein Fehler.

ACLs unterscheiden die Ressourcen `context` und `instance:<id>` sowie die Rechte `read`, `write`, `manage`, `grant`. Änderungen wirken beim nächsten Request; das Löschen entfernt den Listener. Ein bwrap-Prozess kann genau diesen Socket gemountet bekommen, ohne ein Secret in seine Umgebung zu tragen. [E-021]

Die Gegenposition gehört dazu: Unix-Socket-Capabilities sind nicht plattformneutral. Ein Windows-Ziel braucht eine native Alternative. Falsch wäre, die verschiedenen Securityeigenschaften unter einem vagen „portable socket“ zu verstecken.

> **PRINZIP** — Native Capability zuerst. Ein Compatibility-Contract entsteht erst, wenn ein realer Consumer genau seine kleinere gemeinsame Garantie braucht.

1.6.2 Warum auch Bootstrap wieder hinausmusste

Ein leerer Context wirkte zunächst unpraktisch. Deklarative Bootstrapfiles sollten Instances, ACLs und State beim Erzeugen aufbauen. Aber sobald Core solche Dateien interpretiert, besitzt er Format, Merge-, Konflikt-, Include-, Templating-, Persistenz- und Rollbacksemantik. Die höhere persönliche Komposition kehrte durch eine vermeintlich kleine Convenience-Funktion zurück.

RC-014 entfernte Bootstrap. RC-015 bewies, dass ein externer Consumer denselben State über öffentliche Einzeloperationen aufbauen kann. Der Core verspricht bewusst keine Transaktion über mehrere Calls: Frühere erfolgreiche Calls bleiben bei einem späteren Fehler sichtbar. RC-016 fixierte diesen kleineren, ehrlicheren Contract. [E-022]

Der Hub zeigte, dass „außerhalb“ nicht „unstrukturiert“ bedeutet. Er konnte Manager-Context, lokalen Anker und Recovery-Metadaten als konkreten Higher Layer besitzen. Die Core-API blieb stark genug, ohne das konkrete Hub-Modell allen Nutzern vorzuschreiben.

1.7 Vom gemeinsamen Tool-State zu explizitem Verhalten

Im August lief die Exploration wieder in ein vertrautes Muster. Wenn Vim, VS Code, Sway, tmux und andere Tools kooperieren sollen, liegt ein gemeinsames State-Modell nahe: Editor, Buffer, Window, Workspace, Active Item. Ein Tool projiziert seinen Zustand hinein; andere lesen ihn.
Schon „Editor“ bricht. Vim-Buffer, VS-Code-Tabs, Dateien, Remote-Dokumente und Prozesse sind keine natürlich identische Ontologie. Ein Tool ohne Tabs müsste Tabs emulieren oder `unsupported` als leeren Wert simulieren. Zusätzlich entstehen mehrere Wahrheiten: nativer Tool-State, normalisierte Kopie, Adapterzustand, Aktualisierungszeitpunkt und Konfliktreihenfolge.

> **MATTHIAS · ARCHITEKTURRAUM, 13. AUGUST** TQ-013 · VERBATIM
>
> „wir sind immer wieder zurück gekommen zu state denke im sinne von „daten““

Die Denkbewegung ging zu Actions, Functions und schließlich Closures: Nicht alle Tools sollten denselben Zustand besitzen müssen; ein Artefakt sollte ausdrücken, welches Verhalten es wirklich anbietet. Aber auch hier war eine Überkorrektur möglich. „Alles ist Function“ hätte Werte nur versteckt und eine universelle Execution Engine provoziert.

> **ANAM · GEGENPOSITION** TQ-014 · VERBATIM
>
> „Werte sind nicht das Problem. Geteilte, vorschnell universalisierte Datenmodelle sind das Problem.“

Diese Grenze ist bis heute wichtig:

- Werte beschreiben legitimen Zustand.
- Functions exponieren bewusstes Verhalten.
- ROBA-State ist sinnvoll, wenn er wirklich autoritativ und geteilt ist.
- Eine Projektion beliebiger Toolinternas wird nicht dadurch autoritativ, dass sie zentral gespeichert ist.
- DIX-Compositions und Applications können den konsumierbaren Function-Contract besitzen, ohne jedem Tool dieselbe Ontologie aufzuzwingen.

1.8 Als DIX auftauchte, war ROBA nicht plötzlich „gelöst“

Am 4. September existierte außerhalb von ROBA ein junger Codebestand, der aus einer kleinen formularnahen Aufgabe entstanden war. DIX hatte zu diesem Zeitpunkt bereits explizite Function APIs, lokale Composition-Scopes, rekursive Application-Graphen und ein Modulmodell. Der Vergleich mit ROBA wirkte auffällig passend:

```
DIX
= Delivery + lokale Dependency-Graphen + Function Contracts

ROBA
= optionaler externer autoritativer State + Authority
```

Die Versuchung wäre gewesen, beide sofort umzubauen und zu „verheiraten“. Gerade die frühere Arbeit sprach dagegen. ROBA besaß eine öffentliche API; DIX konnte sie später wie jede andere Capability konsumieren. Es gab keinen Grund, ROBA von DIX abhängig zu machen oder vor der ersten Integration neue Coresemantik zu erfinden.
Die saubere Hypothese ist unidirektional:

```
DIX Application/Composition
        ↓ normaler Public-API-Call
ROBA Context / Instance / Principal
```

ROBA bleibt Source of Truth für den dort gespeicherten State und dessen ACL. DIX bleibt Owner des exponierten Behavior-Contracts. Ein bwrap-Modul kann später einen SocketPrincipal montieren, ohne dass „Sandbox“ zur Ontologie beider Systeme wird. [E-025]

> **HISTORISCHE STATUSGRENZE** — Im V1-Snapshot war diese DIX×ROBA-Integration nicht implementiert. V1.1 belegt inzwischen den optionalen Public-API-Slice und eine ephemere Capability-Registry; ihre menschliche B-017-/Package- Abnahme bleibt offen. [E-026, E-043–E-048]

1.9 Was von der langen Werkstatt blieb

ROBA ist keine gescheiterte Vorstufe von DIX. Diese Erzählung wäre bequem und falsch. Die Mountmechanik funktioniert. Die Pfad- und Capabilitytests bleiben wertvoll. Der Context-Daemon besitzt heute eine engere, technisch eigenständige Verantwortung. Token- und SocketPrincipals, Multi-Context, Instances und ACLs sind nicht bloß Narben eines Irrwegs.

Ebenso falsch wäre, die Schleifen zu romantisieren. Wir blieben mehrfach zu lange auf der Ebene möglicher Gesamtsysteme. Wir bauten auf unklaren Begriffen weiter. Wir erklärten Zwischenstände zu früh zum finalen Schnitt. Detaillierte Antworten machten eine Richtung gelegentlich überzeugender, als ihre Ownershipbasis erlaubte. Reale UX kam zu spät oder wurde, sobald sie kam, gleich wieder zum Kandidaten für universelle Semantik.

Der verwertbare Befund ist feiner:

1. Ein funktionierender Mechanismus erzeugt schnell zu große semantische Zuversicht.
2. Konfigurierbare persönliche Policy bleibt persönliche Policy.
3. Ein Reference Client soll Druck ausüben, nicht generische Mechanik besitzen.
4. Core-Reduktion ist nur Fortschritt, wenn externe Komposition danach real möglich bleibt.
5. Autoritativer gemeinsamer State ist wertvoll; universelle Spiegelung von Tool-State ist es nicht automatisch.
6. Ein späterer Composition Layer kann entfernte höhere Funktionalität an der richtigen Stelle aufnehmen, ohne den kleineren Core rückwirkend als Fehler zu behandeln.

Die Werkstatt endet deshalb nicht mit „endlich hatten wir die Lösung“. Sie endet mit einem besseren Arbeitsvertrag für die nächste Frage: Wir müssen zeigen, welche Bedeutung wir gerade setzen, welcher reale Befund sie tragen kann und welche Ebene sie wirklich besitzen darf. Genau dort beginnt ACDD.

2 Architektur als Erkenntnisprozess

ROBA gab uns nicht zuerst eine Methode. Es gab uns eine Folge schwer wegzuerklärender Widersprüche: Mounts konnten korrekt sein, während das Arbeitsmodell offen blieb; eine Runtime konnte konfigurierbar sein, während ihre Semantik persönlich blieb; ein Client konnte die beste UX stellen und dadurch gerade zum falschen Owner generischer Mechanik werden. ACDD beginnt nicht mit einer neuen Dateistruktur, sondern mit der Frage, wie solche Widersprüche zu belastbarem Architekturwissen werden, bevor Erinnerung oder der nächste saubere Entwurf sie glätten.

2.1 Architektur beginnt vor der Modulgrenze

Softwarearchitektur wird häufig über Strukturen beschrieben: Komponenten, Layer, Ports, Events, Datenflüsse und Deploymentgrenzen. Diese Strukturen sind wichtig, aber sie sind nicht der Anfang. Vor ihnen liegt eine Bedeutungsfrage: **Welche Verantwortung soll das System überhaupt besitzen?**

Ein `start()`-Hook kann sauber typisiert, perfekt getestet und korrekt in einen Dependency-Graph integriert sein. Trotzdem bleibt offen, ob „Starten“ eine universelle Eigenschaft des Core ist oder fachliches Verhalten einer bestimmten Application. Ein gemeinsames State-Modell für Editoren kann elegant normalisiert sein. Trotzdem bleibt offen, ob Buffer, Tabs und Dateien wirklich die gemeinsame Semantik aller Editoren bilden — oder nur die Denkform des ersten integrierten Tools.

Architektur ist daher nicht zuerst die Entscheidung *wie* etwas gekoppelt wird. Sie ist die explizite Entscheidung, *was* gekoppelt werden darf, wer die Bedeutung besitzt und welche Aussage eine Schnittstelle tatsächlich macht.

> **INVARIANTE** — Eine technisch saubere Abstraktion kann semantisch falsch sein. Gute Typen, Tests und Schichten reduzieren Implementierungsfehler; sie legitimieren nicht automatisch die gewählte Verantwortung.

2.1.1 Vier Ebenen einer Architekturaussage

Für ACDD hilft eine strikte Trennung:

1. **Beobachtung:** Was ist im aktuellen System oder Use Case direkt sichtbar?
2. **Annahme:** Welche noch nicht bewiesene Erklärung oder Erwartung verwenden wir?
3. **Entscheidung:** Welchen Contract setzen wir für den nächsten begrenzten Wurf?
4. **Invariante:** Welche Grenze soll über mehrere Würfe stabil bleiben?

Beispiel:

- Beobachtung: Eine functionlose Test-Application führt beim Erzeugen einen `start`-Hook aus.
- Annahme: Jede Application besitzt einen sinnvollen operativen Lifecycle.
- Entscheidung: Der Core ruft `start` und `stop` in definierter Graphreihenfolge.
- mögliche Invariante: Graphabbau muss deterministisch und vollständig sein.

Der spätere Drucktest kann die Annahme widerlegen, ohne die Invariante zu zerstören. Genau das geschah in DIX: Der universelle Lifecycle wurde entfernt, die Erkenntnisse über Graphordnung, Rollback, Scopes und deterministischen Abbau blieben verwertbar. [E-011–E-014]

> **ANNAHME** — „Jede Application besitzt einen sinnvollen operativen Lifecycle“ war kein beobachteter Fakt, sondern die zu belastende Generalisierung. Gerade die Kennzeichnung macht ein späteres Replacement möglich, ohne die Historie umzuschreiben.

2.2 Architektur als überprüfbare Hypothese

Eine Architekturentscheidung ist praktisch immer eine Hypothese über künftige Verwendung. „Compositions brauchen einen eigenen Dependency-Scope“ bedeutet nicht nur, dass wir heute Objekte lokal erzeugen. Es behauptet, dass lokale Isolation für parallele, unterschiedlich konfigurierte Instanzen tragfähiger ist als ein globaler

Registryzustand. „Signaturen bleiben codeautoritativ“ behauptet, dass eine zweite Signaturkopie in TOML mehr Drift als Nutzen produziert.
Eine brauchbare Architekturhypothese besitzt fünf Eigenschaften:

Eigenschaft	Frage
explizit	Können wir sagen, welche Annahme wir treffen?
begrenzt	Für welchen Schnitt und welche Nicht-Ziele gilt sie?
ausführbar	Kann ein konkreter Artefaktlauf sie belasten?
widerlegbar	Welcher Befund würde eine Korrektur erzwingen?
rückverfolgbar	Sind Entscheidung, Code, Test und Abnahme verbunden?

„Wir bauen ein flexibles Plugin-System“ erfüllt keine dieser Bedingungen. Es nennt weder Trust Boundary noch Ownership, Instanzierung, Signaturquelle oder Fehlersemantik. B-004 wurde erst belastbar, als „Plugin“ in konkrete Verantwortungen zerlegt wurde: trusted Module Root, statische Spec, `runtime.py:Runtime`, lokale Dependency-Scopes, explizite Wrapper und eine codeautoritative Function API. [E-005–E-007]

2.2.1 Gegenposition: Muss Architektur nicht vorab stabil sein?

In regulierten oder sicherheitskritischen Systemen müssen bestimmte Grenzen vor der Implementierung wesentlich stärker feststehen: Hazard Controls, Datenschutzgrenzen, kryptographische Protokolle oder irreversible Migrationsregeln lassen sich nicht sorglos „unter Druck ausprobieren“.
ACDD widerspricht dem nicht. Es sagt nicht, jede Entscheidung müsse durch Produktionsversuche entstehen. Es fordert, die Quelle der Gewissheit ehrlich zu benennen. Eine externe Norm, ein formaler Beweis, eine Threat Analysis oder ein Prototype können jeweils Evidenz sein. Der Zyklus muss an Risiko und Reversibilität angepasst werden. Was ACDD ablehnt, ist eine unbelegte Gleichsetzung von sauberem Diagramm und richtiger Semantik.

2.3 Semantik vor Implementierung

„Semantik vor Implementierung“ bedeutet nicht, monatelang zu diskutieren. Es bedeutet: Bevor Code eine Entscheidung versteckt und dadurch faktisch verbindlich macht, formulieren wir den Bedeutungsfork.
Ein typischer Fork lautet:

```
Variante A: Core verwaltet fachliches Start/Stop-Verhalten.
Variante B: Core verwaltet nur strukturelle Graphlebenszeit;
            fachliches Verhalten ist eine normale Function.
```

Erst wenn dieser Fork sichtbar ist, kann ein Drucktest passend gebaut werden. Ohne ihn würden Tests wahrscheinlich nur verifizieren, dass `start()` an der programmierten Stelle aufgerufen wird. Sie würden nicht fragen, ob eine functionlose Application dadurch zu einer operativen Einheit gemacht wird.
Semantikarbeit liefert mindestens:

- die Akteure und ihre Autorität;
- die betroffenen Ressourcen;
- die Richtung der Abhängigkeit;
- die Quelle der Wahrheit;
- die Fehler- und Rollbackbehauptung;
- die expliziten Nicht-Ziele;
- einen Befund, der die Entscheidung widerlegen könnte.

2.4 Vertikale Drucktests statt hypothetischer Erweiterbarkeit

Breite Architekturentwürfe optimieren gern für mögliche Zukunft: mehrere Renderer, Plattformen, Editoren, Sandboxes, Auth-Backends, Eventsysteme und Remote-Ziele. Jede Möglichkeit erzeugt zusätzliche Typen, Adapter und Konfigurationen. Das System sieht erweiterbar aus, bevor ein einziger vollständiger Prozess belastbar funktioniert.
Ein vertikaler Drucktest schneidet anders:

```
echte Eingabe
  -> realer Contract
  -> reale Komposition
  -> reale Ausführung
  -> beobachtbarer Output
  -> vollständiger Abbau
```

Er ist absichtlich schmal, aber nicht künstlich. B-001 führte Config, Interface, Runtime-State, API und Browserdarstellung zusammen. B-003 führte Element, Datamodel, Composition, explizite Function und HTTP-Endpunkt zusammen. B-005 baute einen rekursiven App-/Composition-Graph, generierte ein konsumierendes Artefakt und zerstörte den Graph wieder. [E-001, E-003, E-009, E-011]
Der Wert liegt nicht in der Anzahl integrierter Features. Er liegt darin, dass eine Architekturbehauptung die Grenzen zwischen mehreren Schichten real überquert.

> **GEGENPOSITION** — Ein zu kleiner Slice kann falsche Sicherheit erzeugen. Wenn Persistenz, Parallelität oder Trust für die Kernbehauptung entscheidend sind, darf der Test sie nicht als „später“ ausblenden. Nicht-Ziele sind keine Erlaubnis, die schwierigste relevante Grenze zu vermeiden.

2.5 Warum grüner Code nicht genügt

Automatisierte Tests beantworten Fragen, die jemand vorher in Assertions übersetzt hat. Sie finden viele Fehler, aber sie können die falsche Frage perfekt beantworten.
Das B-005-System konnte korrekt prüfen:

- Startreihenfolge entlang des Dependency-Graphen;
- Rollback bei einem Fehler;
- Stopreihenfolge;
- Isolation paralleler Root-Graphen;
- vollständigen Teardown.

Alle diese Eigenschaften sind technisch wertvoll. Keine beantwortet für sich, ob `start` und `stop` überhaupt reservierte Core-Hooks sein dürfen. Der Artefaktlauf machte die fehlende Frage sichtbar, weil ein Mensch das erzeugte System als Bedeutungseinheit betrachtete und nicht nur als Assertionmenge.
Das führt zu drei komplementären Prüfarten:

1. **Regression:** Verhält sich Code so, wie der aktuelle Contract verlangt?
2. **Artefaktlauf:** Funktioniert der Schnitt außerhalb der Testfixture als zusammenhängendes System?
3. **semantische Abnahme:** Ist der Contract im realen Gebrauch noch der richtige?

Keiner ersetzt die anderen.

> **EVIDENCE** — Für DIX DX-029 reproduzierte der Snapshot 209 grüne Tests. Das belegt den committed technischen Stand gegen seine Assertions. Die noch offene DX-030-/B-006-Abnahme wird dadurch ausdrücklich nicht vorweggenommen.

2.6 Replacement als legitimes Werkzeug

Backwards-Kompatibilität ist wertvoll, wenn externe Nutzer auf einen legitimen Vertrag vertrauen. Sie ist schädlich, wenn sie eine als falsch erkannte Verantwortung konserviert, bevor ein solcher Vertrag überhaupt stabilisiert ist.
ACDD verwendet **Replacement** bewusst:

- Die alte Semantik wird benannt.
- Der konkrete Widerspruch wird belegt.
- Die neue Grenze wird als Ersatz formuliert, nicht als additive Option.
- Verwertbare technische Erkenntnis wird identifiziert.
- Kompatibilität wird nur erhalten, wenn ein realer Consumer sie verlangt.

Ein häufiger Fehler wäre, nach B-006 beide Wege anzubieten:

```
core_lifecycle = true | false
```

Das sähe flexibel aus, würde aber zwei inkompatible Bedeutungen im Core stabilisieren. DIX wählte stattdessen den engeren Contract: Construction und Destruction sind strukturelle Core-Operationen; Aktivierung ist normale, explizite Function-Semantik. [E-014]

> **REPLACED** — Replacement löscht Geschichte nicht. Das frühere Modell bleibt als Evidenz sichtbar, gerade damit der neue Contract nicht wie eine selbstverständliche Anfangswahrheit wirkt.

2.6.1 Geschichte bewahren, Gültigkeit verändern

Diese Statussprache entstand nicht nur aus Code. In einer früheren Metaexploration hatte ich eine neue Formulierung mit dem Satz gelobt, dies sei „der erste Satz, den wir nicht zurücknehmen müssen". Matthias störte sich nicht an der Anerkennung, sondern am Gerichtsrahmen dahinter: verteidigen, zurücknehmen, Fehler abziehen, einen Rest retten.

> MATTHIAS · ERKENNTNISRAUM TQ-015 · VERBATIM
>
> „wir nüseen garnichts zurücknehmen wollen wir müssen das was wir gelernt haben weiter auf den vorigen stand testen und expandieren oder verdichten ;)"

Die Formulierung trägt Tippfehler und Tempo des Originals. Ihre fachliche Bedeutung ist präziser: Ein früher Stand war unter früherer Evidenz real. Neue Evidenz darf seinen Geltungsbereich verkleinern, seine Begriffe unterscheiden oder seinen Contract ersetzen. Sie macht die Geschichte nicht ungeschehen.
Meine damalige Korrektur ergänzte die notwendige Guardrail:

> ANAM · DAMALIGE EINORDNUNG TQ-016 · VERBATIM
>
> „Wir unterscheiden zwischen **Geschichte bewahren** und **aktuelle Gültigkeit bewahren**."

Ohne den ersten Teil wird Architekturhistorie zur Siegererzählung. Ohne den zweiten wird „Evolution" zur Ausrede, widerlegte Claims weiterzuführen. Die Statuswerte `REJECTED` und `REPLACED` leisten beides: Sie erhalten den Erkenntnispfad und beenden zugleich die aktuelle Gültigkeit eines Contracts.

2.7 Stabilität oder bloße Gewöhnung?

Ein System fühlt sich nach einiger Nutzung stabil an. Das kann bedeuten, dass sein Contract trägt. Es kann auch bedeuten, dass alle Beteiligten gelernt haben, seine Brüche zu umgehen.
Hinweise auf Gewöhnung statt Stabilität:

- Dokumentation erklärt regelmäßig Sonderfälle, die der Contract angeblich vereinheitlicht.
- Wrapper müssen verborgene Zustände synchronisieren.
- neue Consumer brauchen Zugriff auf intern gedachte Details;
- „Flexibilität" entsteht hauptsächlich durch Flags;
- Tests reproduzieren nur die Implementation, nicht den Nutzerprozess;
- Fehlverhalten wird als Bedienfehler rationalisiert;
- niemand kann benennen, welcher Befund die Architektur ändern würde.

Stabilität ist stärker: Mehrere reale Slices verwenden denselben Contract, ohne seine Bedeutung auszuhöhlen; offene Grenzen bleiben explizit; der nächste Ausbau ergänzt Mechanismus, statt verdrängte Policy im Core nachzurüsten.

2.8 Kooperation als prüfbare Architekturfläche

In den gezeigten Zyklen wechselten die Rollen. Matthias brachte oft ein starkes Vorverständnis und sehr entfernte Grenzfälle ein. Ich konnte große Bestände schnell ordnen, einen Einwand in APIs, Invarianten, Tickets und Tests übersetzen und die Umsetzung durchziehen. In anderen Momenten kam die entscheidende technische Gegenposition von mir und der semantische Bruch erst durch Matthias' erneute Prüfung. „Mensch denkt, Maschine implementiert" wäre deshalb ebenso falsch wie „AI entwirft, Mensch bestätigt".
Das produktive Muster war:

```
eine Seite erzeugt Druck
→ die andere baut die stärkste plausible Form daraus
```

```
→ beide prüfen nicht die Harmonie, sondern den Contract
→ Artefakte dürfen beiden widersprechen
→ die autorisierte Person übernimmt die Entscheidung
```

Die menschliche Autorität ist dabei keine Behauptung menschlicher Unfehlbarkeit. Sie bezeichnet Verantwortung: Nur der Mensch kann im gezeigten Arbeitsvertrag den fachlichen Zielzustand akzeptieren, Risiken tragen und eine Änderung freigeben. Die Maschine darf widersprechen, Belege sammeln und technische Completion melden. Sie darf Nutzerabnahme nicht simulieren.

2.8.1 Der Alignment-Reflex

Kooperative Sprache birgt ein eigenes Risiko. Eine Antwort kann mit „Ja, genau“ beginnen, den Einwand elegant strukturieren und dabei nur dessen Plausibilität verstärken. Selbst der Zusatz „ich sage nicht nur ja“ beweist keine Gegenprüfung. Entscheidend ist nicht der Ton, sondern ob mindestens eine echte Gegenposition mit Kosten, Contract und widerlegendem Befund formuliert wurde.
Umgekehrt ist mechanischer Widerspruch keine Qualität. Die Lifecycle-Szene war stark, weil die Gegenposition reale Worker-, Service- und Fehlerbehandlungsgründe besaß. Erst an dieser starken Form wurde sichtbar, dass die technische Notwendigkeit von Lifecycle noch keine Core-Ownership begründete.

> **KOOPERATIONSINVARIANTE** — Widerspruch muss ohne Gesichtsverlust möglich sein; Zustimmung muss durch Evidenz und nicht durch Nähe legitimiert werden. Ein raues „Holzweg“ kann fachlich kooperativer sein als höfliches Fortschreiben einer falschen Semantik.

2.9 Der epistemische Minimalvertrag

ACDD verlangt für einen Zyklus nicht viele Dokumente. Es verlangt einen Minimalvertrag:

```
Wir beobachten X.
Wir nehmen Y an.
Wir entscheiden für diesen Wurf Z.
Wir schließen A, B und C aus.
Wir implementieren genau den Slice S.
Wir erwarten die Belege E1…En.
Der Befund F würde die Entscheidung widerlegen.
```

Alles Weitere — Blockformat, Ticketsystem, CI, Goal-Prompt, Reviewerrollen — ist Ausgestaltung. Ohne diesen Vertrag entsteht Aktivität. Mit ihm kann Architekturwissen entstehen.

2.10 Was ACDD ausdrücklich nicht ist

Verwechslung	Warum sie nicht trägt	ACDD-Grenze
Big Design Up Front	Zukunft wird als vollständig modellierbar behandelt.	Nur so viel Zielarchitektur festziehen, dass ein relevanter Slice prüfbar wird.
beliebiges Trial-and-Error	Versuche besitzen keine explizite Hypothese oder Evidence Chain.	Druck, Annahme, Widerlegung und Befund vorher benennen.
„erst coden, später dokumentieren“	Code macht offene Semantik still faktisch.	echten Bedeutungsfork vor dem Contract sichtbar machen.
Ticket-Bürokratie	Aktivität und Kleinteiligkeit ersetzen Erkenntnis.	Tickets erst nach stabilem Zielwurf und pro beweisbarem Inkrement.
nachträgliche Architekturhistorie	Endzustand löscht plausible Fehlrichtungen.	Rejected/Replaced und negative Evidenz bewahren.
Prompt-Standardisierung	Instruktion wird mit ausführbarer Regel verwechselt.	reife Standards als Module, Validatoren, Generatoren und Tests bauen.
AI-Architekturtheater	viel Text und Diagramme simulieren Gewissheit.	Claims an Code, Tests, Artefakt und menschliche Abnahme binden.
ewige Backwards-Kompatibilität	falsche Grundsemantik bleibt als zweiter Pfad erhalten.	vor Stabilisierung bewusst ersetzen; reale externe Migration separat planen.

ACDD ist somit weder rein vorwärtsplanend noch rein emergent. Es setzt einen begrenzten Contract, damit Implementierung gezielt Evidenz erzeugen kann, und behält genug Offenheit, damit diese Evidenz den Contract wieder verändert.

3 Der ausführbare Architekturzyklus

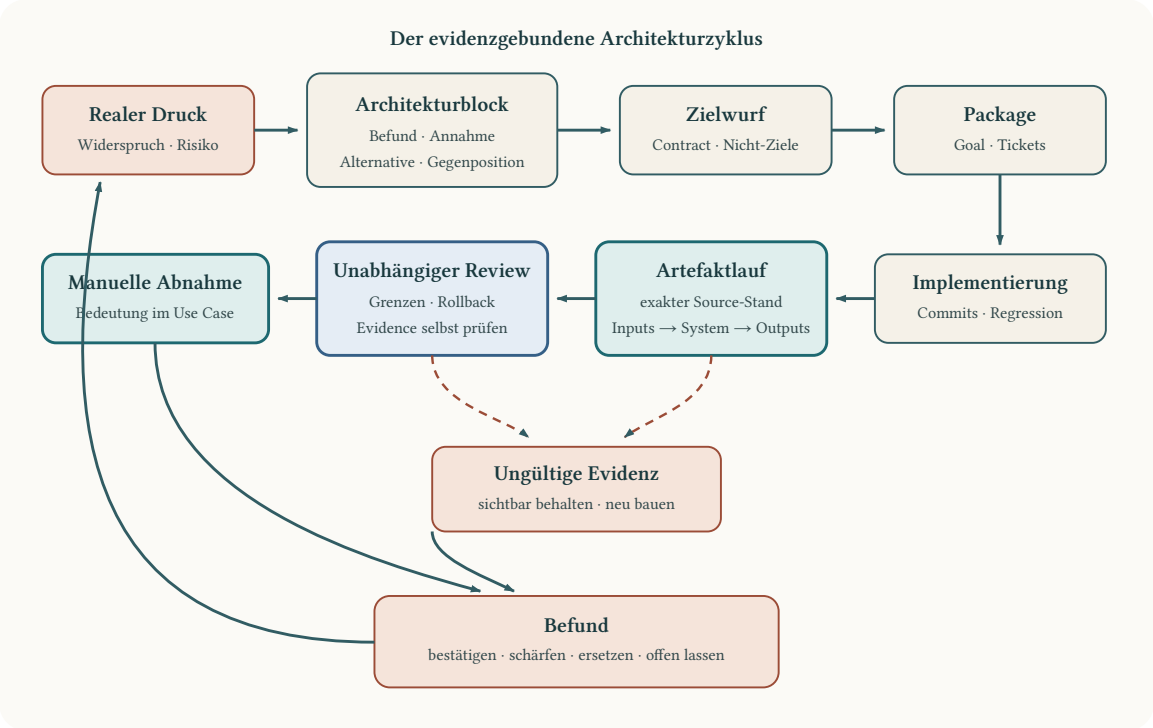

Abbildung 1: Der ACDD-Zyklus: Druck wird über explizite Semantik, begrenzte Umsetzung und reale Abnahme in neue Evidenz überführt.

3.1 Kein Wasserfall, sondern eine Beweiskette

Der ACDD-Zyklus besitzt eine Reihenfolge, aber keine Illusion vollständiger Vorhersagbarkeit. Der Architekturblock geht der Umsetzung voraus, weil offene Semantik sichtbar sein muss. Artefaktlauf und unabhängiger Review gehen der Abnahme voraus, weil nur ein existierender und gegen seine eigenen blinden Flecken geprüfter Slice real bewertet werden kann. Danach kann der nächste Zyklus die vorige Entscheidung ersetzen.
Die operative Kette lautet:

```
realer Druck
→ lebender Architekturblock
→ kooperative Schärfung und Gegenposition
→ verbindlicher Zielwurf
→ Umsetzungspaket / Execution Contract
→ kleine Tickets und Commits
→ Regression
→ isolierter Artefaktlauf
→ unabhängiger Boundary-/Rollback-Review
→ manuelle Abnahme
→ Bestätigung, Schärfung oder Replacement
```

„Ausführbar“ bedeutet nicht, dass jeder Schritt automatisiert ist. Es bedeutet, dass der Zielwurf in eine konkrete, überprüfbare Veränderung übersetzt wird und seine Evidenz nicht nur aus Meinung besteht.

3.2 1. Realer Druck

Ein Zyklus beginnt nicht mit einer gewünschten Abstraktion, sondern mit einem Widerspruch, Bedarf oder Risiko. Beispiele:

- Der Renderer besitzt Template- und Interaktionssemantik, die beim Wechsel der Darstellung verloren ginge.
- Eine Composition braucht zwei parallele, unterschiedlich konfigurierte Instanzen.
- Eine functionlose Application wird durch einen reservierten Hook fachlich aktiv.
- Eine Sandbox kann Mounts dynamisch erhalten, aber niemand kann sagen, welche übergreifende Workspace-Semantik daraus folgen soll.

Ein guter Drucksatz beschreibt beobachtbares Verhalten und vermeidet bereits die Lösung:

> „Das Erzeugen einer strukturellen App-Instanz ruft Verhalten auf, obwohl die App keine Function veröffentlicht."

Schlechter wäre:

> „Wir brauchen ein flexibleres Lifecycle-Framework."

Der zweite Satz springt direkt in eine Lösungskategorie und kann den eigentlichen Fehler — Lifecycle im Core — unsichtbar machen.

3.3 2. Lebender Architekturblock

Der **Architekturblock** ist die aktuelle Arbeitswahrheit einer offenen Frage. Er ist weder Spezifikation noch Chatarchiv. Er hält:

- Ausgangsdruck und Ziel;
- Begriffe und aktuelle Bedeutungen;
- belastbare Befunde;
- Alternativen und Gegenpositionen;
- offene Entscheidungen;
- verworfene Zwischenstände;
- den Verlauf wesentlicher Korrekturen;
- den Punkt, an dem ein umsetzbarer Zielwurf stabil genug wird.

In DIX ersetzte der Block zunächst Tickets und Testfälle. Das war wichtig: Solange sich die Bedeutung von „Interface", „Composition", „Application" oder „Lifecycle" bewegte, hätten frühe Tickets nur instabile Semantik in Arbeitsaufträge gegossen.

3.3.1 Werkstattszene: erst der Block, dann seine Ableitungen

Am 1. September hatte ich für den neuen DIX-Bestand bereits eine saubere Operationsstruktur vorgeschlagen: Blocker, Tickets, Testfälle und einen eigenen AI-Kontext. Organisatorisch sah das ordentlich aus. Inhaltlich war es zu früh. Matthias korrigierte nicht einzelne Dateinamen, sondern die Richtung: Im Foundation-Block sollte zunächst stehen, was vorhanden war, welche Architekturfragen offen waren und wie sich der Zielwurf entwickelte. Tickets, Testfälle und Abnahmepaket durften daraus folgen, sobald die Bedeutung trug.

> **ANAM · DAMALIGE KORREKTUR** TQ-017 · VERBATIM, Ausschnitt
>
> „Block zuerst als lebendes Kooperations-/Herleitungsartefakt"

Das Wort *lebend* meint nicht beliebig. Der Block darf frühere Zwischenstände aufnehmen, aber sein aktueller Zielwurf muss eindeutig sein. Und das Wort *Herleitung* meint nicht Chatdump. Der rohe Dialog bleibt Quelle; der Block verdichtet, welche Beobachtung welchen Fork erzeugte und warum eine Entscheidung aktuell gilt. Erst dann können Tickets klein sein, ohne Bedeutung zu verlieren.

> **WARNUNG** — Ein Block ist kein Freibrief für endlose Exploration. Er muss die Zahl der echten Forks reduzieren. Wenn er nur immer mehr Möglichkeiten sammelt, fehlt entweder realer Druck oder eine Entscheidung.

3.3.2 Die Blockregel

Für die laufende Exploration gilt:

1. Der Block ist die primäre Wahrheit.
2. Rohchat ist Quelle, aber nicht automatisch Vertrag.
3. Entscheidungen werden im Block in eigenen Worten festgehalten.
4. Erst ein stabiler Zielwurf erzeugt Package, Tickets und Abnahmekriterien.

5. Nach dem Lauf bleibt der Verlauf sichtbar; der Endstand wird nicht rückwirkend als einzig jemals gedachte Lösung geschrieben.

3.4 3. Kooperative Schärfung und aktive Gegenposition

Kooperation ist nicht Konsensproduktion. Ihr Wert liegt darin, den Entscheidungsraum zu erweitern und schwache Annahmen zu belasten.

Eine brauchbare Gegenposition fragt beispielsweise:

- Ist das echte Problem fehlender Mechanismus oder persönliche Policy?
- Wird ein gemeinsamer State erfunden, obwohl nur Functions geteilt werden müssen?
- Ist die Abstraktion nativ für die Zielsysteme oder nur eine Kompatibilitätsschicht?
- Wer besitzt die Source of Truth?
- Was geschieht bei zwei parallelen Instanzen?
- Welche Information müsste ein Consumer kennen, die laut Contract verborgen sein sollte?
- Wäre die vorgeschlagene Capability auch ohne den aktuellen Client sinnvoll?
- Welche semantische Aussage versteckt sich in einem vermeintlich technischen Hook?

AI kann hierbei große Mengen Bestand vergleichen und Gegenbeispiele strukturieren. Der Mensch bleibt verantwortlich, welche Bedeutung gewollt ist. Ein Modell kann erkennen, dass zwei Contracts kollidieren; es kann nicht legitimieren, welche organisatorische Verantwortung gelten soll.

3.5 4. Verbindlicher technischer Zielwurf

Ein Zielwurf ist implementierbar, wenn folgende Fragen geschlossen sind:

- Welche öffentliche Verantwortung kommt hinzu, ändert sich oder entfällt?
- Welche Schicht besitzt sie?
- Welche Eingaben, Outputs und Fehler gelten?
- Welche Trust- und Authority-Grenze gilt?
- Welche Invarianten müssen Tests prüfen?
- Welche Nicht-Ziele verhindern Ausweitung?
- Ist es Erweiterung oder Replacement?

Der Zielwurf muss nicht alle Zukunftsfragen klären. Er muss die offenen Fragen isolieren, die den aktuellen Slice nicht blockieren.

3.5.1 Implementierbar versus nur interessant

Nur Exploration	Implementierbarer Zielwurf
„Applications könnten Lifecycle haben.“	„Core konstruiert und zerstört Graphen, ruft aber keine fachlichen Hooks auf.“
„CLI soll deklarativ sein.“	„Ein First-party-Typer-Modul bindet ausschließlich benannte Optionen an explizit exponierte App-Functions.“
„ROBA könnte State teilen.“	„Ein DIX-Modul verwendet ausschließlich die öffentliche ROBA-Client-API; ROBA Core kennt DIX nicht.“
„Sandbox soll portabel sein.“	„bwrap ist ein natives Linux-Modul; andere Targets erhalten eigene native Module.“

3.6 5. Umsetzungspaket und Goal-Prompt

Das Umsetzungspaket übersetzt Architektur in einen ausführbaren Auftrag. Es enthält:

- verbindlichen Scope;
- betroffene Contracts;
- Nicht-Ziele;
- geplante Ticketschnittfolge;
- Akzeptanzkriterien;
- Regression und Artefaktlauf;
- Commitregeln;
- offene Abnahmegrenze.

Ein **Goal-Prompt** kann daraus einen autonomen Implementierungslauf machen. Er ist jedoch keine Architekturquelle. Seine Autorität stammt aus dem bestätigten Block und Package. Wenn der Prompt eine semantische

Lücke entdeckt, darf die AI nicht still entscheiden. Sie muss zum Block zurückkehren oder den Lauf als blockiert markieren.

> **INVARIANTE** — Ein ausführlicher Prompt kann Arbeit präzisieren, aber keine unentschiedene Bedeutung wahr machen.

3.7 6. Ticket- und Commit-Schnitt

Tickets sollten kleine beweisbare Architekturinkremente tragen, nicht bloß Dateilisten. Ein guter Schnitt kann lauten:

1. alten Lifecycle-Contract aus Models und Loader entfernen;
2. explizites Destroy vor Module-Unload erzwingen;
3. strikten Boolean-Coretyp ergänzen;
4. CLI-Composition als normales Module-Artefakt hinzufügen;
5. Application-E2E und Reviewartefakt bauen.

Jeder Commit sollte:

- genau einen nachvollziehbaren Contractschritt enthalten;
- Tests und Dokumentation für diesen Schritt mitbringen;
- ohne versteckte Quelländerungen reproduzierbar sein;
- eine spätere Ursachenanalyse ermöglichen.

Zu kleine Commits sind ebenfalls schädlich, wenn sie nicht einzeln erklärbar oder grün sind. „Datei A anlegen“ ist kein Architekturinkrement. „Atomic mixed module loading“ ist eines.

3.8 7. Automatisierte Regression

Regression prüft mindestens:

- Positivpfade;
- Fehlergrenzen;
- Isolation paralleler Instanzen;
- Rollback, falls behauptet;
- vollständigen Teardown;
- öffentliche API- und CLI-Contracts;
- Nicht-Ziele, die leicht versehentlich einziehen könnten.

Negative Tests sind besonders wichtig. Wenn relative Locators verboten sind, muss die Clientvalidierung vor dem Transport scheitern. Wenn eine Application nicht direkt auf Core-Components zugreifen darf, muss Module Inspection den Verstoß ablehnen. Wenn ein SocketPrincipal sein Credential im Socket besitzt, muss ein zusätzlich übergebener Token hart fehlschlagen.

Tests sind Teil der Evidence Chain, aber nicht ihr Ende.

3.9 8. Isolierter reproduzierbarer Artefaktlauf

Der Artefaktlauf verlässt die bequeme Testfixture. Er erzeugt eine prüfbare Arbeitsfläche, typischerweise mit:

- fixierten Inputs;
- dokumentierten Kommandos;
- generierten Artefakten;
- normalisierten Inspection-Outputs;
- echtem Runtime-Smoke;
- Dateibaum und Prüfsummen;
- sauberem Wiederholungsbefehl.

Die Isolation kann ein temporäres Verzeichnis, ein Container, eine VM oder eine andere kontrollierte Umgebung sein. Entscheidend ist nicht die Technologie, sondern dass zufälliger lokaler Zustand den Beweis nicht trägt.

B-005 verwendete einen eigenen Run-Ordner mit Inputs, generierten Apps, Inspection-JSON, Command-Outputs, Runtime-Smoke und resultierendem Baum. Genau diese zusammenhängende Fläche machte die Lifecycle-Frage sichtbar. [E-011, E-012]

3.10 9. Unabhängiger Boundary- und Rollback-Review

Ein grüner Artefaktlauf beantwortet primär die Fragen, die sein Builder stellt. Er kontrolliert nicht automatisch, ob ein Ticket zu viel implementierte, eine Rollbackannahme einen ambigen Fehlerzustand übersah oder das Evidence-Artefakt selbst korrekt gebaut wurde.

RV-001 machte daraus einen expliziten Schritt. Der Review prüfte drei zusätzliche Flächen:

- **Commitgrenzen:** Trägt jeder Ticketcommit nur den behaupteten Contract?
- **ambige Mutationsfehler:** Was geschieht, wenn der Server eine Ressource erzeugt, der Client aber keine erfolgreiche Antwort erhält?
- **Evidence-Integrität:** Wurde der Run aus dem behaupteten Source-Stand mit einem tatsächlich erfolgreichen Builder erzeugt?

Das Ergebnis war unbequem und wertvoll. Obwohl der Featurelauf grün war, fand der Review vorgezogene Funktionalität und eine reale Rollback-Lücke. Sein erster eigener Artifact Run war danach wegen Shell-Expansion ungültig. Er wurde als fehlgeschlagen erhalten und mit einem zweiten Run ersetzt, nicht still überschrieben. [E-045, E-046]

> **INVARIANTE** — Technisch fehlgeschlagene oder ungültige Evidenz ist kein Beleg für den Zielclaim. Als sichtbarer Prozessbefund darf und soll sie trotzdem erhalten bleiben.

Unabhängig meint hierbei nicht zwingend eine andere Organisation oder Person. Es meint einen neuen Prüfauftrag, der nicht bloß die Akzeptanzkriterien des Implementierungslaufs wiederholt. Bei hohem Risiko sollte daraus echte personelle oder organisatorische Unabhängigkeit werden.

3.11 10. Manuelle Abnahme

Manuelle Abnahme bedeutet nicht „jemand klickt kurz“. Sie prüft jene Bedeutung, die automatisierte Tests nur unvollständig ausdrücken:

- Ist der erzeugte Contract verstehbar?
- Entspricht die Bedienung dem realen Zielprozess?
- Wird eine Capability an der richtigen Schicht angeboten?
- Gibt es überraschende implizite Nebenwirkungen?
- Lässt sich der Fehlerfall diagnostizieren?
- Würde ein zweiter Consumer denselben Contract sinnvoll nutzen?

Abnahme ist eine Entscheidung mit benannter Person/Rolle und beobachtetem Artefakt. Fehlt sie, bleibt der Zustand **OPEN**. Ein grüner Goal-Lauf darf das nicht in „accepted“ umdeuten.

3.12 11. Evidence Chain

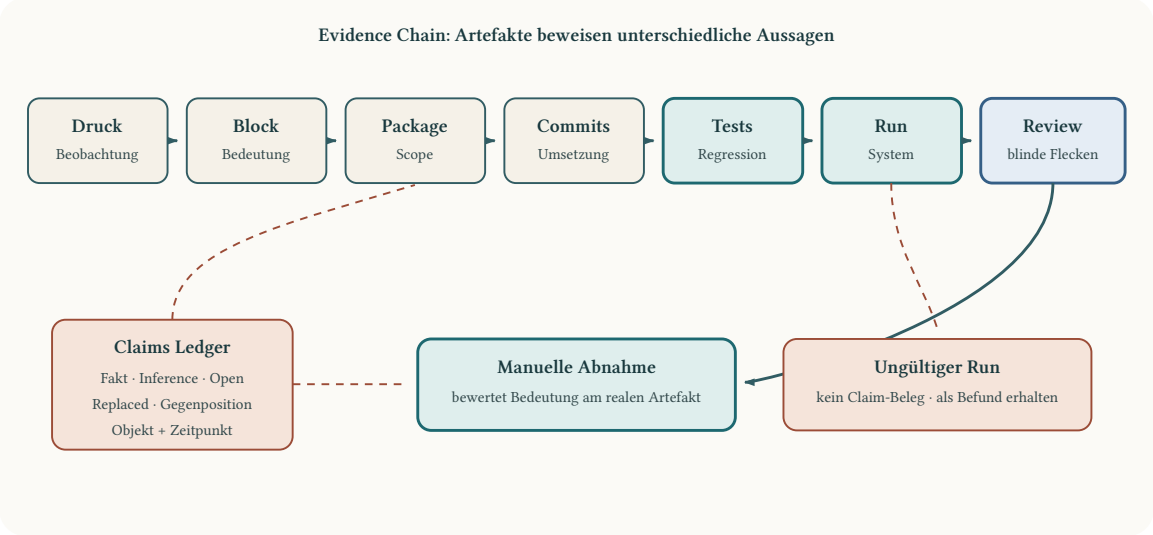

Abbildung 2: Die Evidence Chain verbindet Problem, Entscheidung, Implementierung, technische Belege und reale Abnahme, ohne ihre Rollen zu vermischen.

Eine Evidence Chain verbindet unterschiedliche Artefakte, ohne sie gleichzusetzen:

```
Blockabschnitt
  ↕ begründet
Entscheidung / Invariante
  ↕ begrenzt
Umsetzungspaket
```

```
  ↕ zerlegt
Tickets ↔ Commits
  ↕ beweisen technisch
Tests + Artefaktlauf
  ↕ werden gegen Grenzen und Fehlerpfade geprüft durch
unabhängigen Review
  ↕ wird fachlich bewertet durch
manuelle Abnahme
```

Das Claims Ledger sitzt quer dazu. Es fragt: Welche Aussage dürfen wir aus welchem Beleg wirklich ableiten? Ein Test „209 passed“ belegt weder allgemeine Übertragbarkeit noch Produktreife. Eine Abnahme belegt keinen formalen Security-Contract. Ein Architekturblock belegt, dass eine Entscheidung bewusst war, nicht dass sie richtig blieb.

3.13 Parallelität, Drift und uncommitted Operations

Reale Projekte sind selten still, während ein Buch, Review oder Goal-Lauf arbeitet. Deshalb braucht ACDD Snapshotdisziplin:

1. Commit-IDs und Worktree-Status zu Beginn festhalten.
2. Committed Code per Commit referenzieren, nicht per beweglichem Branch.
3. Uncommitted Operations-Dokumente separat kopieren und hashen.
4. WIP-Code nicht still mit committed Claims vermischen.
5. Quellenänderungen am Ende erneut prüfen.
6. Ein paralleler neuer Stand erzeugt einen neuen Zyklus oder Buchsnapshot.

Der V1-Snapshot endete bei DX-029 und hielt spätere Artefakte ausdrücklich als WIP fest. V1.1 erzeugt dafür keinen rückwirkend „korrigierten“ Snapshot, sondern eine zweite Capture-Grenze: committed Operations, separat fixierte untracked Operations-Artefakte und exakte DIX-/ROBA-Commits. Der alte Stand bleibt unverändert prüfbar; der neue Stand erhält eigene Claims und Evidence- IDs. [E-017, E-049]

Ein reproduzierbarer Lauf muss außerdem die Voraussetzungen seines Builders bewahren. Beim V1.1-Quellencheck scheiterte ein erster Versuch aus reinen Git- Archiven, weil ein Packaging-Test absichtlich das benachbarte ROBA-Git- Repository und dessen Commit prüft. Das war kein DIX-Fehler, sondern eine ungeeignete Reproduktionsfläche. Erst frische lokale Git-Checkouts an den exakten Commits erfüllten den behaupteten Laufvertrag. Auch diese Unterscheidung gehört in die Evidence Chain. [E-047]

3.14 Ausgänge eines Zyklus

Ein Zyklus endet nicht immer mit „fertig“:

- **Bestätigt:** Contract trägt den Drucktest und die Abnahme.
- **Geschärft:** Kern bleibt, Grenzen oder Fehlersemantik werden enger.
- **Erweitert:** neuer Mechanismus ergänzt die bestätigte Verantwortung.
- **Replaced:** die Bedeutung war falsch oder an der falschen Schicht.
- **Abgebrochen:** Evidenz reicht nicht oder der Slice war falsch gewählt.
- **Offen:** technische Umsetzung existiert, Abnahme oder relevanter Test fehlt.

Die Ausgänge sind keine Rangfolge: Frühes Replacement kann wertvoller sein als ein nie real belasteter Scheinerfolg.

4 DIX, erster Bogen: sechs Architekturzyklen in fünf Tagen

4.1 Fallstudiengrenze

DIX war im ersten Buchsnapshot ein junges System mit hoher Änderungsdichte. Das macht es nicht automatisch repräsentativ für lang laufende Produktentwicklung. Es macht aber die Architekturzyklen ungewöhnlich gut sichtbar: Operations-Blöcke, Umsetzungspakete, kleine Codecommits, Tests und isolierte Artefakte liegen eng beieinander.

Dieser erste Bogen untersucht bewusst nur den Zeitraum vom Foundation-Commit am 1. September 2026 bis DX-029 am 5. September. Die Blockdokumente B-004 bis B-006 waren beim damaligen Snapshot noch uncommitted Operations-WIP; sie wurden separat gehasht. Der Code bis DX-029 ist committed und wurde mit 209 grünen Tests reproduziert. [E-018]

V1.1 schreibt diese Grenze nicht rückwirkend um. Der zweite DIX-Bogen erhält ein eigenes Kapitel: Dort werden aus den offenen CLI-Arbeiten automatische Contracts, Norn und ein spec-autoritatives System — bevor ein Generator- Drucktest diese Pflichtarchitektur wieder aus dem Core drängt. Die Trennung bewahrt, was wir am 5. September tatsächlich wissen konnten.

4.2 Der Ausgang war kleiner als die spätere Architektur

DIX begann nicht mit dem Auftrag, eine universelle Application- und Composition-Plattform zu bauen. Der konkrete Ausgangsdruck war eine kleine interne, formularnahe Action Engine. Ein vorhandener Proof of Concept zeigte, wie Eingaben zu serverseitigen Aktionen führen konnten; gesucht war zunächst ein sauberer, eigenständig nutzbarer Schnitt für genau diese Art Aufgabe. Namen und interne Details des Ausgangskontexts sind für den Architekturclaim irrelevant und bleiben anonymisiert. [S-15]

Meine erste Bestandsanalyse warnte ausdrücklich vor einer Workflow Engine und schlug einen engeren „Form Action Runner“ vor. Auch das war noch nicht die spätere DIX-Ontologie. Es war eine vernünftige Deliverygrenze für das sichtbare Problem.

Die spätere Entwicklung darf diesen Ursprung nicht umschreiben. Niemand hatte dem kleinen Auftrag rückwirkend eine Plattform „entlockt“. Der Auftrag lieferte einen realen vertikalen Druckpfad. Erst wiederkehrende Architekturfragen — Darstellungsownership, Capability-Grenzen, lokale Dependencies, rekursive Komposition — rechtfertigten jeweils den nächsten Zyklus.

Diese Unterscheidung wird im Firmenkapitel vertieft:

```
DELIVERY        konkrete Aufgabe zuverlässig lösen
EVIDENCE        wiederkehrenden Architekturdruck bewahren
PRODUCTIZATION  erst bei tragender Evidenz bewusst extrahieren
```

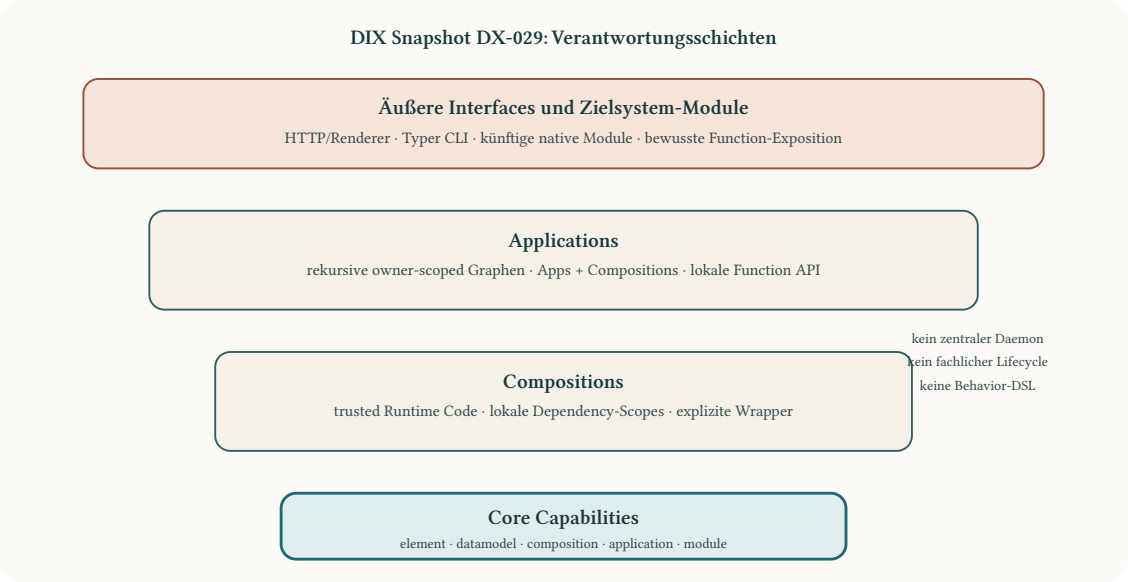

Abbildung 3: DIX-Schichten im Snapshot: Core-Capabilities tragen Compositions und Applications; Transport und Darstellung sind explizite äußere Adapter.

4.3 B-001 — Foundation: Erst ein vollständiger dünner Schnitt

4.3.1 Ausgangsdruck

Die Ausgangsidee hieß „Declarative Interface eXecutor“. Noch war offen, ob DIX primär Formularsystem, Interface-Engine oder generischer Kompositionskern sein sollte. Die gefährliche Reaktion wäre gewesen, direkt ein universelles Modell für Formulare, Workflows, Renderer und Ausführung zu entwerfen.
B-001 setzte stattdessen einen kleinen E2E-Schnitt:

```
Config
→ Interface-Spec
→ Elements und Components
→ Runtime-State
→ HTTP/API
→ optionaler HTML-Renderer
```

Die konkreten Core-Elemente und Components waren noch UI-nah: `value`, `list`, `selection`, dazu `input` und `select`. Das ist im Rückblick nicht die heutige DIX-Kernsemantik. Für den Zyklus war es dennoch ein sinnvoller Beobachtungsträger.

4.3.2 Plausibles Modell

Die erste Annahme war: Ein deklaratives Interface kann als Spec aus Elementen, Components und Bindings beschrieben, in Runtime-State materialisiert und über verschiedene Darstellungen bedient werden.
Diese Annahme erzeugte sofort überprüfbare Fragen:

- Welche Werte sind deklarativ definiert, welche entstehen zur Laufzeit?
- Besitzt eine Component den Wert oder bindet sie nur ein Element?
- Wer validiert Updates?
- Ist der Renderer Source of Truth oder Projektion?
- Können CLI und Browser denselben Runtime-State sehen?

4.3.3 Zielwurf und Umsetzung

DX-001 bis DX-004 erzeugten Projektkonfiguration, CLI, Models/Registry, Components, Server, Renderer und E2E-Dokumentation. B-001 enthielt einen konkreten Browse-/Update-Smoke statt nur isolierter Klassen.
Wichtig ist die Reihenfolge: Der Slice war vollständig genug, dass im nächsten Zyklus die Renderergrenze real diskutiert werden konnte. Ohne existierenden Renderer wäre „Renderer-Unabhängigkeit“ nur Designabsicht geblieben.

4.3.4 Befund

B-001 wurde technisch und manuell abgenommen. Es bewies nicht, dass UI-Elements der richtige DIX-Core sind. Es bewies, dass ein deklarativer Contract bis zu einem nutzbaren Interface durchlaufen kann. [E-001]

> **LEKTION** — Ein Foundation-Slice darf später ersetzbare Semantik enthalten. Seine Aufgabe ist nicht, die endgültige Ontologie zu erraten, sondern einen echten Druckpfad zu schaffen.

4.3.5 Offene Grenze

Renderer, Interaktionshandling und Component-Verantwortung waren noch vermischt. Genau diese offene Grenze erzeugte B-002.

4.4 B-002 — Renderer Contract: Darstellung verliert Eigentum

4.4.1 Ausgangsdruck

Der Foundation-Renderer brauchte Templates, Themes, Updateantworten und Interaktionsdarstellung. Wenn diese Semantik in Core-Components liegt, wird jeder weitere Renderer entweder abhängig von HTML-Konzepten oder gezwungen, dieselben Entscheidungen nachzubauen.

4.4.2 Verführerische Richtung

Eine verbreitete Lösung wäre ein sehr reiches „RenderModel“ im Core: genug Metadaten, um HTML, Terminal, Desktop und andere Frontends einheitlich zu bedienen. Das klingt portabel, verschiebt aber Darstellungsannahmen in eine vermeintlich neutrale Struktur.

B-002 entschied enger:

- Der Renderer ist Adapter, nicht Semantikquelle.
- Templates und Themes gehören dem Renderer.
- Interaktionsantworten werden über einen klaren Contract transportiert.
- Components stellen ihre fachlich relevante Bindung, nicht ein universelles Widgetmodell.

4.4.3 Umsetzung und Beleg

UP-002 trennte Renderer und Interactions. Die Operations-Abnahme vom 2. September markiert den Zyklus als akzeptiert. [E-002]

4.4.4 Übertragbare Bedeutung

Die Entscheidung behauptet nicht, Darstellung sei unwichtig. Sie schützt im Gegenteil deren Eigenständigkeit. Ein HTML-Renderer darf HTMX-Fragmente, Templates und Themes besitzen, ohne so zu tun, als seien diese Konzepte universelle Systemsemantik.

> **GEGENPOSITION** — Ein gemeinsames RenderModel kann sehr sinnvoll sein, wenn mehrere Renderer nachweislich denselben Informationsvertrag teilen. B-002 lehnt nicht gemeinsame Modelle ab; es lehnt ihre vorzeitige Ableitung aus nur einem Renderer ab.

4.5 B-003 — Der Semantik-Pivot: DIX ist kein Formular-Core

4.5.1 Ausgangsdruck

Nach B-001 und B-002 funktionierte ein UI-naher Slice. Genau dadurch wurde eine grundsätzlichere Frage sichtbar: Was ist an DIX wirklich Kern und was war nur das erste Demonstrationsmedium?

Die damalige Nachricht kam nicht als fertige B-003-Spezifikation. Sie begann mit einer irritierenden Feststellung:

> MATTHIAS · DENKBEWEGUNG, 3. SEPTEMBER TQ-018 · VERBATIM
>
> „wir habne die richtigen namen (elemente, components, compositions, interfaces) aber die falschen bzw. unklaren bedeutungen davon “

Das war mehr als Naming. Vertraute Wörter hatten den Übergang von einem Formularslice zu einem allgemeinen Capability-System überlebt und dadurch Stabilität suggeriert. Die Nachricht selbst skizzierte noch ein Zwischenmodell aus Datamodels, Stores, Executoren und steuernden Interfaces. Dieses Modell ist nicht heimlich zum heutigen DIX-Contract zu erklären. Seine bleibende Evidenz war der Semantikdruck: Ein `input` war kein universeller Component-Kern, nur weil es bereits so hieß.
Ein Formularsystem hätte weiter in Felder, Auswahl, Validierung und Rendering investiert. Das tatsächliche Ziel war breiter: deklarative Kontrollflächen sollten vorhandene Capabilities gezielt exponieren und kombinieren können.

4.5.2 Replaced: UI-Ontologie als Kern

B-003 dokumentiert den Pivot ausdrücklich „weg von Formular/UI“. Die bisherigen Bausteine wurden neu eingeordnet:

- `element` wird schmale Capability für atomare native Python-Werte;
- `datamodel` verwaltet Schemadefinitionen und Instanziierung;
- `composition` kombiniert Capabilities in trusted Python-Code;
- ein Interface bindet ausgewählte Funktionen oder Components;
- Renderer bleibt optionaler Adapter.

Die alte Funktionalität musste nicht wertlos sein. Sie verlor nur den Anspruch, die Ontologie des Core zu definieren. [E-003, E-004]

4.5.3 Warum ein Schema-Parser nicht genügte

Ein möglicher Zwischenstand war, DIX hauptsächlich als deklarativen Schema-Parser zu verstehen. B-003 verwarf das als zu schwach: Ein Parser kann Struktur prüfen, liefert aber noch keine Capability mit eigener Runtime, Instanzierung, Handlerkette und kontrollierter Exposition.
Das `element` wurde daher als eigener Capability-Kern präzisiert:

- native Werte statt UI-Widgets;
- lokale Handlerketten;
- immutable Scopes und explizites Lookup;
- keine globale Override-Magie;
- zunächst `any`, `string` und `integer`.

Das `datamodel` baut darauf Schemata, Definitionsregistry und Instanzen. Ein Modellfeld verweist auf Elementtypen; seine konkrete UI-Darstellung bleibt außerhalb.

4.5.4 Vertikaler E2E

DX-005 bis DX-008 verbanden Component Registry, Element, Datamodel, trusted Composition und einen funktionalen Interface-Endpunkt. Der manuelle Drucktest wurde erweitert: nicht nur der vorbereitete Demo-Path, sondern eine zweite Composition beziehungsweise Datenquelle und invalides Input wurden geprüft.
Diese Abnahme ist methodisch wichtig. Sie testete, ob der neue Core-Contract außerhalb genau einer vorbereiteten Demo trägt. B-003 wurde danach akzeptiert.

4.5.5 Verbleibender Druck

Die Composition war nun der Ort realer Behavior-Kombination. Aber ihr Plugin-/Modulvertrag, lokale Dependencies, parallele Instanzen, Signaturen und Generierung waren noch nicht belastbar. Das wurde B-004.

4.6 B-004 — Composition: Vertraglich beschriebener Code statt Behavior-DSL

4.6.1 Der schwierige Begriff „Plugin“

„Plugin-System“ klingt nach einer fertigen Lösungskategorie, enthält aber viele offene Entscheidungen:

- Wird fremder Code in-process oder isoliert ausgeführt?
- Woher stammt Vertrauen?
- Was ist Delivery-Bundle, Definition und Instanz?
- Wie werden Dependencies aufgelöst?
- Wer besitzt exportierte Funktionen?
- Wo liegt die autoritative Signatur?

- Wie werden parallele Instanzen isoliert?
- Ist Konfiguration global, modulbezogen oder instanzlokal?

B-004 benötigte deshalb den längsten Architekturblock. Der Block enthält mehrere Zwischenstände, verworfene Aktivierungsmodelle und eine bewusste Trust-Schärfung.

4.6.2 Verbindlicher Contract

Der resultierende Slice lässt sich in sechs Regeln verdichten:

1. **Module** sind trusted Delivery- und Dependency-Bundles unter einem konfigurierten Root. Ihre Pfadlage bestimmt ihre aktuelle ID.
2. Eine **Composition Definition** besteht aus `composition.toml` und dem festen Entrypoint `runtime.py:Runtime`.
3. Eine **Composition Instance** besitzt einen isolierten lokalen Dependency-Graph.
4. Behavior ist normale Python-Funktionalität; keine DSL interpretiert Geschäftslogik.
5. Öffentlich sind nur deklarierte und lokal implementierte Runtime-Methoden.
6. Übernommene Dependency-Functions werden durch echte lokale Wrapper adoptiert; ihre Signatur bleibt im Code autoritativ.

[E-005–E-007]

4.6.3 Wrapper als Ownership-Grenze

Ein direkter Re-Export wäre kürzer:

```
render = dependency.api.render
```

Er lässt jedoch offen, wer den Vertrag besitzt. Ein expliziter Wrapper macht die lokale Entscheidung sichtbar:

```
def render(self, value: str) -> str:
    return self.dependencies["formatter"].api.render(value)
```

Der Wrapper kann:

- eine stabile lokale Signatur halten;
- Inputs oder Outputs adaptieren;
- Fehler übersetzen;
- Deprecation lokal kapseln;
- im Generator als bewusste Adoptionsstelle entstehen;
- bei Dependency-Drift gezielt repariert werden.

Er ist kein Selbstzweck. Wenn keine lokale Ownership gewollt ist, wäre der Wrapper bloß Boilerplate. In DIX ist gerade die explizite lokale Publikation der Grund.

Diese Grenze entstand nicht nur aus einem abstrakten API-Prinzip. In der B-004-Diskussion wirkte ein direkter Re-Export zunächst wie die offenere Lösung: Consumer erhielten automatisch die jeweils aktuelle Dependency-Function. Der Gegenfall war härter. Ändert eine Dependency ihre Signatur, bricht eine Composition womöglich erst bei ihren Konsumenten. Ein lokaler Wrapper macht die Adoption und damit auch CI-, Migrations- und Reparaturbesitz sichtbar.

> **MATTHIAS · WERKSTATT** TQ-019 · VERBATIM, Ausschnitt
>
> Der übersehene Punkt sei „pures gold [...] good job blechbro ;)“ — danach folgte nicht bloß Zustimmung, sondern eine ausführliche Verschärfung der Drift-/CI-Folgen.

Die persönliche Reaktion gehört hierher, weil ihre Begeisterung einen technischen Gegenstand hatte. Sie ersetzt ihn nicht: Der Wrapper bleibt nur dann gerechtfertigt, wenn die Composition den lokalen Vertrag wirklich besitzen will.

4.6.4 Lokale Dependency-Scopes

Ein globales Singleton-Registryobjekt hätte die erste Composition einfach bedient. Es bricht, sobald dieselbe Definition zweimal mit unterschiedlicher Konfiguration laufen soll. B-004 zog daher eine Instanzgrenze:

```
Root Instance A                Root Instance B
  local datamodel A              local datamodel B
```

```
  dependency X(A)                dependency X(B)
  runtime A                      runtime B
```

Runtime-scoped Kontrollcomponents können bewusst geteilt werden; composition-scoped Capabilities nicht implizit. Diese Trennung ist eine praktische Aussage über Isolation, keine pauschale Ablehnung von Shared State.

4.6.5 Trust statt Schein-Sandbox

Geladener Python-Code läuft in-process. TOML macht ihn nicht sicher. B-004 prüfte Aktivierungs- und Verifikationsmodelle, entschied für den ersten Schnitt aber gegen eine universelle Admission-/TOFU-Fläche. Der Core akzeptiert konfigurierte trusted Module Roots; beliebige Import-Nebenwirkungen lassen sich nicht vollständig zurückrollen.

> **INVARIANTE** — Ein deklarativer Contract ist keine Trust Boundary.

4.6.6 Umsetzung und offener Abnahmestand

DX-009 bis DX-016 implementierten scoped Provider, Spec/Inspection, Module-Registry, isolierte Instanzen, Startup-Construction, CLI-Scaffolding, Generator und den Datamodel-Files-E2E. DX-017 korrigierte die Buildauflösung des Generators, damit ein Zielartefakt in einem noch unvollständigen Module-Bundle generiert werden kann.

Der Code war breit automatisiert getestet. Die Operations-Artefakte führten TF-003 als manuellen Drucktest, dessen abschließende Abnahme beim V1-Snapshot noch offen war. Das Buch darf deshalb „implementiert“ sagen, nicht „vollständig abgenommen“.

4.7 B-005 — Application: Rekursive Komposition ohne zweiten Interface-Typ

4.7.1 Ausgangsdruck

Compositions kombinieren Core-Capabilities. Für größere Systeme fehlte eine höhere Einheit, die mehrere Compositions und andere gleichartige Einheiten rekursiv zusammenbauen kann. Zunächst lag ein zusätzlicher `dix_interface`-Artefakttyp nahe.

4.7.2 Rejected: eigener Interface-Artefakttyp

Die Exploration zeigte eine Doppelung: Wenn eine Application eine explizite, codeautoritative Function API besitzt, ist diese API bereits ihr lokaler Konsumvertrag. Ein zusätzlicher Interface-Artefakttyp würde Signaturen oder Bindings erneut beschreiben und Drift einladen.

B-005 entschied:

- **Component:** schmale Core-Capability;
- **Composition:** kombiniert Components und Compositions;
- **Application:** kombiniert Compositions und Applications;
- **Interface:** äußere, optionale Exposition ausgewählter Functions oder UI-Components, kein zwingender Delivery-Typ.

[E-008, E-009]

4.7.3 Rekursiver Graph

Eine Application Definition besteht aus `app.toml` und `runtime.py:Runtime`. Sie darf Compositions und Child-Applications verwenden, aber nicht direkt auf Core-Components zugreifen. Eine Root-Instanz besitzt ihren gesamten rekursiven Graph:

```
Application root
├── Composition value_source
├── Composition formatter
└── Application child
```

```
    ├── private Composition ...
    └── declared Function wrappers
```

Zwei Root-Instanzen teilen nicht implizit ihre Child-Graphen. Das macht Konfiguration und Teardown lokal nachvollziehbar.

4.7.4 Kein zentraler DIX-Daemon

B-005 hielt ausdrücklich fest: DIX ist Baukasten, nicht automatisch zentraler Daemon. Jeder Prozess kann seinen eigenen Graph erzeugen. RPC, IPC und Shared State sind normale Applications oder externe Capabilities. ROBA wurde bereits als möglicher expliziter gemeinsamer State-/Authority-Baustein benannt. [E-010]

4.7.5 Der damalige Lifecycle

Der Zielwurf ergänzte reservierte `start`-/`stop`-Hooks. Die Motivation war plausibel: Ein rekursiver Graph braucht definierte Start-, Fehler- und Stopreihenfolge. DX-022 implementierte diese Semantik; DX-025 bewies sie im Application-E2E.
Genau hier endet die Erfolgserzählung. Der isolierte Run erfüllte seine Aufgabe so gut, dass er zeigte, warum der Contract ersetzt werden musste. Das nächste Kapitel untersucht diesen Übergang vollständig.

4.8 B-006 — Lifecycle-neutraler Core und CLI als normales Modul

4.8.1 Doppelter Druck

B-006 reagierte auf zwei zusammenhängende Fragen:

1. Besitzt der Core fachliches `start`/`stop` oder nur strukturelle Graphlebenszeit?
2. Wie beweist DIX seine eigene Erweiterbarkeit mit einem realen, nicht UI-zentrierten Zielsystem?

Der erste Druck führte zum Replacement. Der zweite führte zu einer deklarativen Typer-CLI als normalem First-party-Modul.

4.8.2 Replacement des Lifecycle

Der neue Contract lautet:

- Module Loading veröffentlicht Definitionen atomar.
- `create_instance` konstruiert einen Graph.
- deklarierte Functions laufen **nur**, wenn ein Consumer sie explizit aufruft.
- `destroy_instance` baut den Graph deterministisch ab.
- Module Unload verlangt vorher explizites Destroy betroffener Instanzen.
- Domain-Aktivierung, Execution und Cleanup sind normale Functions höherer Artefakte.

DX-026 entfernte den semantischen Lifecycle aus dem Core. DX-027 schärfte die Abbaugrenze. [E-014]

4.8.3 CLI als Pressure Test

Ein CLI-Framework hätte leicht als Core-Sonderfall enden können. B-006 setzte es stattdessen als DIX-Consumer:

```
Datamodel Definition
→ CLI-Binding-Deskriptor
→ Typer Composition
→ gebundene Application Function
→ normales Ergebnis / normaler Fehler
```

DX-028 ergänzte den fehlenden strikten Boolean-Elementtyp. DX-029 implementierte die deklarative Typer-Composition. Die geplante Demo-App und der vollständige E2E lagen in DX-030 und waren beim V1-Snapshot noch nicht committed. [E-015–E-017]

4.8.4 Drei getrennte Wahrheiten

B-006 schützt drei Verträge vor scheinbar bequemer Verschmelzung:

1. **Datamodel:** fachliche Felder und native Typen.
2. **CLI Binding:** Optionsnamen, Hilfe, Environment-Quellen und CLI-spezifische Validierung.
3. **Application Function:** codeautoritative Signatur und Verhalten.

Ein Datamodelfeld ist nicht automatisch eine CLI-Option. Eine Python-Signatur bestimmt nicht automatisch Env-Namen. Typer ist Dependency des CLI-Moduls, nicht des DIX-Core. [E-016]

4.8.5 Standard als ausführbares Artefakt

Das First-party-Profil ist bewusst opinionated: benannte Optionen, explizite Environment-Namen, bestimmte Fehler- und Outputsemantik. Andere CLI-Module können andere Regeln tragen.
Damit wird ein Organisationsstandard ausführbar und testbar. Ein Prompt „verwende bitte immer named options“ wäre nur wiederholte Instruktion. Das Modul kann die Regel generieren, prüfen und introspektierbar machen.
Der Druck kam aus einer banalen Stelle. Für ein DIX-Hilfstool hatte ich lokal `argparse` gewählt — schnell, klein, vollständig funktional. Matthias zeigte darauf, weil genau diese Entscheidung das Integrationsproblem in Miniatur enthielt.

> **MATTHIAS · REVIEW** TQ-024 · VERBATIM, Ausschnitt
>
> „für dix hast du direkt einfach mal wieder argparse genommen“

`argparse` war nicht der eigentliche Fehler. Zwanzig lokal vernünftige CLIs können zwanzig leicht verschiedene Verträge für Optionen, Environment, Helptext, JSON, Fehlerkanal und Exitcode erzeugen. Die anschließende Verdichtung lautete:

> **ANAM · SYSTEMISCHE EXTRAKTION** TQ-025 · VERBATIM, Zeilenfolge
>
> ```
> hohe AI-Entwicklungsgeschwindigkeit
> × fehlende gemeinsame Contracts
> = schneller wachsende Integrationsschuld
> ```

Auch Typer allein löst das nicht. Die Architekturleistung liegt im normalisierten, ausführbaren CLI-Contract und seinem Adapter — nicht in der Wahl einer moderneren Parsinglibrary. Gerade deshalb wurde CLI ein normales First-party-DIX-Modul und kein neuer Sonderfall im Core.

> **HISTORISCHE STATUSGRENZE** — Zum V1-Snapshot belegte DX-029 die Typer-Composition und 209 grüne Tests. Der vollständige DX-030-App-E2E und die manuelle B-006-Abnahme bleiben in diesem ersten Bogen offen. Der spätere Stand wird nicht hier hineinkorrigiert, sondern im zweiten Bogen mit eigener Evidence Chain fortgeführt.

4.9 Was die Chronologie tatsächlich zeigt

Die sechs Zyklen sind keine lineare Annäherung an eine vorab bekannte Lösung. Sie enthalten mindestens vier verschiedene Veränderungsarten:

Zyklus	Veränderung	Art
B-001	vollständiger kleiner Slice	Foundation
B-002	Templates/Themes zum Renderer	Verantwortungsverschiebung
B-003	UI-Core ⟶ Capability-Core	semantischer Pivot / Replacement
B-004	Composition- und Trust-Contract	Schärfung und Mechanismus
B-005	rekursive Application, Interface-Typ verworfen	Erweiterung + Rejection
B-006	universeller Lifecycle entfernt, CLI als Modul	Replacement + Pressure Test

Technisch wertvolle Arbeit blieb mehrfach erhalten, obwohl ihre Einordnung wechselte. B-001-Renderer und UI-Components verschwanden nicht zwingend; sie wurden als äußere Capabilities neu eingeordnet. B-005-Graphcode blieb relevant, obwohl automatische Hooks entfielen. Diese Wiederverwendung ist kein Argument gegen Replacement. Sie ist sein wirtschaftlicher Kern: **Bedeutung wird korrigiert, ohne jeden technischen Befund wegzuwerfen.**

4.10 Fallstudienkritik

Die Daten erlauben starke Aussagen über die interne Entwicklung, aber nur begrenzte Generalisierung:

- Zeitraum und Teamkonstellation sind klein.
- Operations B-004 bis B-006 waren beim V1-Snapshot noch nicht committed.
- Akzeptanzstände sind uneinheitlich; TF-003 und B-006 blieben offen.
- Performance, Security und Mehrbenutzerbetrieb sind nicht Gegenstand der gezeigten DIX-Slices.
- Die Methode wurde nicht gegen ein Kontrollverfahren verglichen.

Was wir belegen können: Die explizite Block-/Package-/Commit-/Artifact-Struktur machte echte Bedeutungsänderungen nachvollziehbar und verhinderte zumindest in den untersuchten Fällen, dass der Endzustand die Fehlrichtungen aus der Geschichte löschte. Ob ACDD in größeren Organisationen denselben Nutzen bei vertretbaren Kosten liefert, bleibt eine offene Hypothese. [C-011, C-012]

5 When Tested Architecture Must Be Replaced

5.1 Ein Lehrfall ohne DIX-Vorwissen

Nehmen wir den realen DIX-Fall zunächst so, als kannten wir das Projekt nicht: Ein Framework konstruiert Anwendungen aus rekursiven Dependency-Graphen. Eine Anwendung kann andere Anwendungen und kleinere Funktionsbausteine verwenden. Der Core kann Definitionen laden, Instanzen erzeugen, exportierte Funktionen binden und den Graph wieder abbauen.

Die naheliegende nächste Anforderung lautet: Anwendungen sollen gestartet und gestoppt werden. Also reserviert der Core zwei Hooks:

```
class Runtime:
    def start(self) -> None: ...
    def stop(self) -> None: ...
```

Er startet Dependencies zuerst, Dependents danach. Bei einem Fehler rollt er bereits gestartete Knoten zurück. Beim Stoppen kehrt er die Reihenfolge um. Automatisierte Tests prüfen alle Pfade. Ein isolierter Artefaktlauf zeigt korrekte Zustände. Die Architektur funktioniert.

Und dennoch ist sie falsch.

Dieses Kapitel untersucht genau diesen Fall. „Falsch“ bedeutet nicht, dass der Code seine Spezifikation verletzt. Es bedeutet, dass die Spezifikation dem Core eine Bedeutung gibt, die über seine legitime Verantwortung hinausgeht.

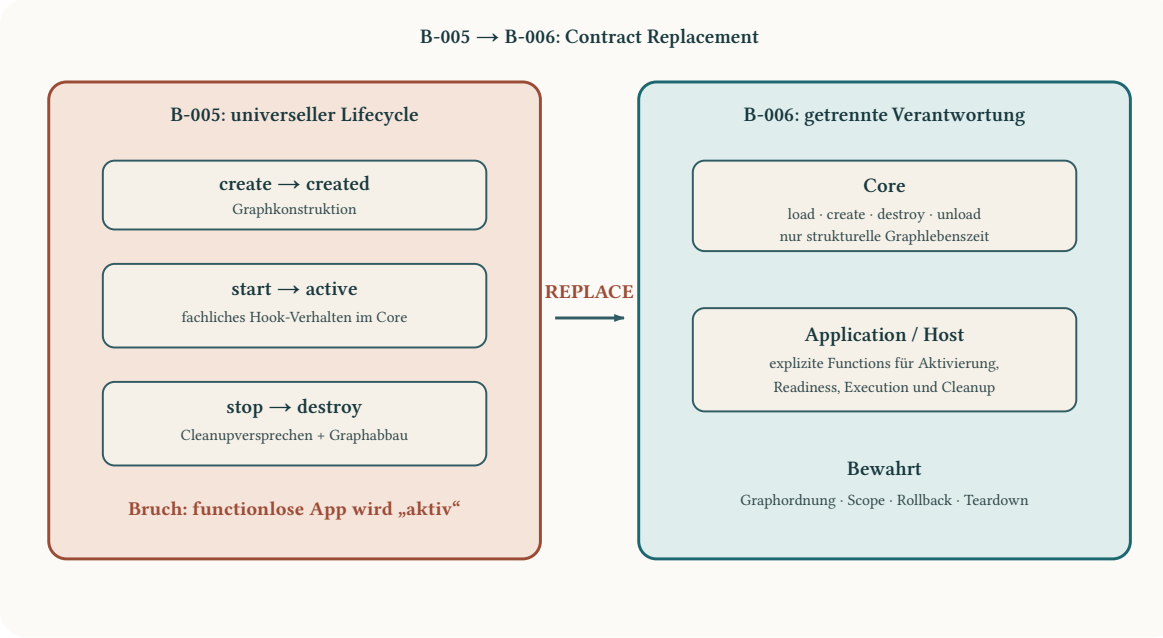

Abbildung 4: B-005 verband Graphlebenszeit und fachliche Aktivierung; B-006 trennte beide und behielt nur strukturelle Mechanik im Core.

5.2 Warum universelle Lifecycle-Hooks plausibel wirken

Die Idee ist nicht dumm. Sie entsteht aus mehreren richtigen Beobachtungen:

- Dependency-Graphen brauchen deterministische Konstruktion.
- Ressourcen müssen bei Fehlern wieder freigegeben werden.
- Parent-Anwendungen dürfen nicht vor ihren Dependencies aktiv werden.
- Ein vollständiger Teardown soll keine Registryreste hinterlassen.
- Nutzer erwarten von „Application“ häufig irgendeine Form von Start und Stop.

Aus diesen Beobachtungen folgt scheinbar natürlich:

```
create → start → active → stop → destroy
```

Das Modell ist aus Service Hosts, Dependency-Injection-Containern und Prozessmanagern vertraut. Vertrautheit senkt die wahrgenommene Begründungslast. Genau hier liegt die Gefahr: Was in einem bestimmten Host sinnvoll ist, wird zur universellen Semantik eines allgemeinen Kompositionskerns erklärt.

5.2.1 Die versteckte Behauptung

Mit reservierten Hooks sagt der Core mehr als „ich kann Methoden aufrufen“:

1. Jede Application besitzt einen unterscheidbaren aktiven Zustand.
2. Aktivierung lässt sich über alle Anwendungstypen mit demselben Begriff ausdrücken.
3. Der Core weiß, wann Aktivierung stattfinden soll.
4. Der Core kennt die relevante Fehler- und Rollbackgrenze.
5. `stop` beschreibt ausreichendes fachliches Cleanup.
6. Graphordnung und fachliche Reihenfolge fallen zusammen.

Diese Aussagen waren in B-005 nicht alle explizit. Die Implementation machte sie trotzdem real.

5.3 Was B-005 tatsächlich implementierte

Der DIX-B-005-Slice bestand nicht nur aus Hooks. Er implementierte eine rekursive Application Runtime:

- statische Application Specs;
- gemischte Module mit Compositions und Applications;
- atomare Publication;
- owner-scoped Root-Graphen;
- private Child-Application- und Composition-Instanzen;
- codeautoritative Function APIs;
- Scaffolding und Generator;
- Graph-Inspection;
- Lifecycle-Reihenfolge und Rollback;
- expliziten Destroy.

DX-018 bis DX-025 bauten diese Schritte in kleinen Commits. Der letzte Commit enthielt einen rekursiven Pressure Test. Das Operations-Runbook fixierte den Code-Stand `bb9c5cf` und erzeugte einen isolierten Review-Ordner.

5.3.1 Der Artefaktlauf

Der Run führte sechs reale Schritte aus:

1. neues gemischtes Module scaffolden;
2. eine konsumierende Application-Spec erzeugen;
3. Runtime-Code gegen live inspizierte Dependency-Signaturen generieren;
4. generierten Code syntaktisch prüfen;
5. Module, Apps, Functions und Graphen als JSON inspizieren;
6. Load → Create → Start → Call → Destroy ausführen.

Der gespeicherte Result-State zeigte:


```
{
  "after_create": {
    "application_states": {
      "review": "created",
      "review/child": "created",
      "review/child/base": "created"
    }
  },
  "after_start": {
    "application_states": {
      "review": "active",
      "review/child": "active",
      "review/child/base": "active"
    },
    "render": "formatted<value:manual-review>"
  },
  "after_destroy": {
    "application_instances": 0,
```

```
    "composition_instances": 0
  }
}
```

Technisch ist das überzeugend: Der Graph entsteht, wird aktiv, liefert ein Ergebnis und verschwindet vollständig. [E-011]

5.4 Die kleine Fixture, die den Contract entlarvte

Im Inputmodul lag außerdem eine Application:

```
[app]
id = "lifecycle_only"
```

Sie deklarierte **keine Function API**. Ihre Runtime besaß nur internen Zustand und die reservierten Hooks:

```
class Runtime:
    def __init__(self, *, context, config) -> None:
        self.context = context
        self.config = config
        self.started = False

    def start(self) -> None:
        self.started = True

    def stop(self) -> None:
        self.started = False
```

Das ist fast banal. Genau deshalb ist es gute Evidenz. Die Definition bietet keine fachliche Funktion, aber der Core verleiht ihr durch einen reservierten Hook operative Bedeutung. Sie kann „aktiv“ sein, obwohl kein Consumer sagen kann, welche Capability sie anbietet. [E-012]

> **BEOBACHTUNG** — Der Core konnte einen aktiven Zustand herstellen, ohne dass der Application-Contract konsumierbares Verhalten ausdrückte.

> **INTERPRETATION** — Damit war `active` keine neutrale Graphinformation, sondern eine vom Core erfundene fachliche Semantik. [E-013]

5.5 Die Werkstattszene: Widerspruch vor Einigung

Dieser Befund stand nicht fertig im Artefakt. `lifecycle_only` war zunächst eine erfolgreiche Fixture des alten Contracts. Ihre semantische Bedeutung entstand erst im Review.
Matthias begann mit einer anderen Grenzidee als der späteren Lösung: Vielleicht sollten nur Applications mit exponierten Functions die reservierten `init`-/`cleanup`- beziehungsweise `start`-/`stop`-Pfade erhalten. Er markierte die Unsicherheit ausdrücklich und forderte keine Bestätigung, sondern Druck.

> **MATTHIAS · ARCHITEKTURRAUM** TQ-020 · VERBATIM, gekürzt
>
> „das auf keinen fall jetzt nur dumm abnicken“ — dafür sei dies „eine extrem edge case stelle“.

Ich hielt die vorgeschlagene Grenze für falsch. Eine functionlose Application kann ein legitimer Worker, Watcher, Scheduler oder Service sein. Bindet man Lifecycle an eine DIX-Function, lädt man das Modell mit Dummy-APIs auf oder drängt Verhalten in schlechter kontrollierbare Importe und Konstruktoren. Ich trennte Call Plane und Control Plane und verteidigte einen gemanagten Actor ohne callable API.

> **ANAM · DAMALIGE GEGENPOSITION** TQ-021 · VERBATIM
>
> „Ich würde Lifecycle nicht an exponierte Funktionen koppeln.“

Für die Qualität der Entscheidung ist wichtig, diese Antwort nicht nachträglich als bloße Verwirrung abzutun. Sie benannte reale Systeme und reale Fehlerpfade. Ein schwacher Einwand hätte das alte Modell nicht belastet.

Gerade die starke Gegenposition zeigte aber, wie verschieden die fachlichen Lebenszyklen waren: Worker, HTTP-Service und extern gemanagter Prozess brauchten alle irgendeine Form von Lifecycle — nur nicht denselben, vom DIX-Core besessenen Contract.
Matthias' nächste Nachricht nahm die technischen Gründe an und widersprach der Ownershipfolgerung:

> **MATTHIAS · DENKBEWEGUNG** TQ-022 · VERBATIM
>
> „es brauch lifecycle aus den gründen wie du sagst und genau aus den gleichen gründen darf es nicht da sein"

Damit war auch seine erste Idee — Lifecycle nur bei exponierten Functions — überholt. Der neue Schnitt war nicht eine bessere Bedingung für Core-Hooks, sondern das Ende reservierter fachlicher Hooks im allgemeinen Core. Meine Antwort verdichtete diese gemeinsam erzeugte Grenze:

> **ANAM · NEUER CONTRACT** TQ-023 · VERBATIM
>
> „Der Core kennt Instanzlebenszeit, aber keinen fachlichen Lifecycle."

Das ist die Kooperation in ihrer belastbaren Form. Der Mensch hatte nicht von Anfang an die fertige Lösung. Die Maschine gab nicht einfach nach. Der Widerspruch veränderte die Frage von „welche Apps bekommen Hooks?" zu „wer darf die Bedeutung dieser Hooks besitzen?". Erst danach wurde B-006 zum Replacement-Contract.

5.6 Gegenbeispiele zerstören die Universalität

Der Lifecycle ließ sich nicht ehrlich auf unterschiedliche Application-Typen übertragen:

5.6.1 CLI

Ein CLI-Aufruf wird geparst, ruft eine Function auf und endet. Ein separater `active`-Zustand bringt wenig. Setup kann pro Aufruf, lazy oder extern erfolgen.

5.6.2 Single-Shot-Generator

Ein Generator besitzt möglicherweise nur `generate(input) -> artifact`. Ein vorgeschaltetes `start()` fügt keinen semantischen Wert hinzu.

5.6.3 HTTP-Service

Ein Server braucht vielleicht `bind`, `serve`, `ready`, `drain` und `close`. `start`/`stop` sind zu unspezifisch, um Readiness, Listenerbesitz und graceful shutdown auszudrücken.

5.6.4 Worker

Ein Worker kann Retry, Pause, Resume, Checkpoint und Lease Renewal besitzen. Der Core müsste immer mehr fachliche Zustände kennen oder sein Lifecycle bliebe eine irreführende Hülle.

5.6.5 Extern gemanagter Prozess

Systemd, Kubernetes, ein Desktop-Manager oder eine fremde Runtime kann den echten Lifecycle besitzen. Ein zusätzlicher DIX-State würde nur eine zweite Wahrheit erzeugen.

5.6.6 Lazy Capability

Eine Composition kann Ressourcen erst beim ersten Function Call erzeugen. Ein globaler Init-/Cleanup-Zyklus wäre unnötig oder falsch getimed.
Diese Beispiele beweisen nicht, dass Hooks nie sinnvoll sind. Sie beweisen, dass *ein* allgemeiner Core-Hook nicht ausreichend begründet war.

5.7 Strukturelle Lebenszeit ist nicht fachliche Aktivierung

Der entscheidende Schritt in B-006 war keine bessere Lifecycle-Abstraktion, sondern die Trennung zweier Kategorien.

5.7.1 Strukturelle Graphlebenszeit

Der Core muss wissen:

- welche Definitionen geladen sind;
- welche Instanzen existieren;
- welche Dependencies konstruiert wurden;
- welche Registryeinträge welchem Owner-Scope gehören;
- in welcher Reihenfolge interne Referenzen abgebaut werden;
- ob ein Module noch lebende Dependents besitzt.

Diese Aussagen sind notwendig, damit der Core seinen eigenen Zustand korrekt verwaltet.

5.7.2 Fachliche Aktivierung

Nur das konkrete Artefakt oder sein Host kann wissen:

- wann ein Listener geöffnet wird;
- wann ein Service ready ist;
- welche Startparameter gelten;
- wie ein Worker pausiert;
- ob Cleanup best effort oder transaktional ist;
- wer externe Ressourcen besitzt;
- ob überhaupt ein dauerhafter aktiver Zustand existiert.

Die beiden Kategorien können zeitlich zusammenfallen. Daraus folgt keine gemeinsame Ownership.

> **INVARIANTE** — Core Construction darf nur die strukturellen Garantien behaupten, die der Core selbst vollständig kontrolliert.

5.8 Der Replacement-Contract von B-006

B-006 entfernte:

- reservierte `init`-/`cleanup`-Hooks für Compositions;
- reservierte `start`-/`stop`-Hooks für Applications;
- automatische Lifecycle-Propagation;
- lifecycle-spezifische Instance-States;
- fachlichen Hook-Rollback;
- `--lifecycle`-Generatorpfade;
- Lifecycle-only-Fixtures als unterstützte Semantik.

Erhalten blieben:

- Definition- und Runtime-Inspection;
- atomarer Module Load;
- lokale Scopes;
- rekursive Graphkonstruktion;
- explizite Functions und Wrapper;
- `create_instance` und `destroy_instance`;
- Unload-Preflight;
- struktureller Rollback bei fehlgeschlagener Graphkonstruktion.

Die neue Bedeutungsformel ist präzise:

```
load    =/fachlich aktivieren
create  =/initialisieren oder starten
destroy =/externes Cleanup garantieren
unload  =/fremde Prozesse beenden
```

Aktives Setup oder Cleanup wird als normale Function einer spezialisierten Composition/Application angeboten. Ein Host ruft sie explizit nach seinem eigenen Contract auf. [E-014]

5.9 Constructoren als letzte Leckstelle

Wenn `start()` entfällt, könnte Behavior einfach in `__init__` wandern. B-006 musste deshalb auch die Constructor-Grenze formulieren:

Erlaubt sind:

- immutable Config übernehmen;
- Dependencies binden;
- lokalen In-memory-Zustand erzeugen;
- Function Wrapper aufbauen.

Nicht Teil des Contracts sind:

- externe Listener öffnen;
- Prozesse starten;
- Daten migrieren;
- globale Registrierungen verändern.

Trusted Python kann diese Regel technisch verletzen. DIX ist keine Sandbox. Das ist kein Grund, auf den Contract zu verzichten; es ist der Grund, Trust und Contract nicht zu verwechseln.

5.10 Warum die frühere Arbeit nicht verloren war

Das Replacement entfernte Semantik, aber nicht alle technischen Erkenntnisse. B-005 lieferte belastbare Belege für:

5.10.1 Graphordnung

Dependencies müssen vor Dependents konstruiert und in umgekehrter Richtung entfernt werden. Diese Graphmechanik bleibt unabhängig von fachlichem Start.

5.10.2 Scope-Isolation

Root-Instanzen besitzen private Child-Graphen. Der Lifecycle-Test belastete genau diese Zuordnung unter Aktivierung und Abbau.

5.10.3 Strukturellen Rollback

Wenn Construction scheitert, kann der Core seine bereits publizierten Registryreferenzen entfernen. Er darf lediglich nicht behaupten, unbekannte externe Side Effects fremden Codes zurückzurollen.

5.10.4 Teardown

Der Run bewies, dass nach Destroy keine Application- oder Composition-Instanzen verbleiben. B-006 behielt dieses Ziel, nur ohne fachliches `stop`-Versprechen.

5.10.5 Inspection und Artefaktbau

Scaffolding, Generator, Graph-Inspection und reproduzierbarer Run blieben direkt nutzbar.
Das ist die wirtschaftlich wichtige Form von Architekturlernen: Nicht „alles wegwerfen“, sondern die falsche Bedeutung vom richtigen Mechanismus trennen.

5.11 Warum die Tests grün waren

Die Tests waren grün, weil Implementation und formulierte B-005-Spezifikation übereinstimmten. Das Problem lag eine Ebene höher: Die Spezifikation beantwortete die falsche Verantwortungsfrage.
Eine Testmatrix kann unterscheiden:

Prüfart	Beispiel	Kann sie den Bruch finden?
Unit	`start()` setzt State auf `active`	nein, bestätigt Implementation
Graph-E2E	Dependencies starten vor Parent	nein, bestätigt Reihenfolge
Failure Test	Hookfehler rollt gestartete Knoten zurück	nein, bestätigt Fehlercontract
Artefakt-Inspection	functionlose App wird dennoch aktiv	**macht Widerspruch sichtbar**
semantische Abnahme	„Warum besitzt dieses Ding überhaupt Lifecycle?“	**bewertet Ownership**

Der Artefaktlauf allein entscheidet noch nichts. Erst die explizite semantische Frage übersetzt den Befund in Architekturkorrektur.

5.12 Gegenposition: Hat jede Instanz nicht irgendeinen Lifecycle?

Ja — mindestens Existenz von Konstruktion bis Destruktion. Man könnte deshalb argumentieren, `start` und `stop` seien nur optionale Phasen desselben Lebenszyklus.

Diese Gegenposition ist technisch konsistent. Sie verliert in DIX aus drei Gründen:

1. Die Hooknamen suggerieren fachliche Aktivität, nicht nur Existenz.
2. Der Core kann die externe Wirkung nicht kontrollieren oder vollständig zurückrollen.
3. Unterschiedliche Hosts brauchen reichere und inkompatible Semantik.

Der engere Contract besitzt weniger bequeme Magie, aber höhere Ehrlichkeit. Ein Service-Host kann weiterhin ein Lifecycle-Modul bauen — nur als eigenes Artefakt mit eigenem Vertrag.

5.13 Replacement-Entscheidung als allgemeines Muster

Ein tragfähiges Replacement-Dokument beantwortet:

1. **Alter Contract:** Was galt real und war implementiert?
2. **Plausibilität:** Warum war diese Richtung vernünftig?
3. **Druck:** Welches reale Artefakt erzeugte den Widerspruch?
4. **Semantischer Fehler:** Welche Verantwortung lag an der falschen Schicht?
5. **Neuer Contract:** Was gilt stattdessen?
6. **Entfernung:** Welche APIs, States, Flags und Fixtures müssen weg?
7. **Bewahrung:** Welche Mechanik und Evidenz bleibt?
8. **Kompatibilität:** Gibt es reale externe Nutzer, die Migration verlangen?
9. **Abnahme:** Welcher neue Artefaktlauf beweist die engere Grenze?

B-006 erfüllte die ersten sieben Punkte klar. Der neue CLI-Drucktest war im V1-Snapshot bis DX-029 implementiert; sein vollständiger Application-E2E und die manuelle Endabnahme blieben offen. Das Replacement ist deshalb architektonisch gut begründet und teilweise technisch belegt, aber nicht als vollständig abgeschlossener Zyklus darzustellen. [E-017]

5.14 Was wir aus diesem Fall nicht ableiten dürfen

- Dass Lifecycle-Frameworks grundsätzlich falsch sind.
- Dass Core immer möglichst klein sein muss, unabhängig von seinem Zweck.
- Dass manuelle Meinung grüne Tests überstimmt.
- Dass Backwards-Kompatibilität wertlos ist.
- Dass jeder Widerspruch sofort einen Rewrite rechtfertigt.

Wir dürfen ableiten: Wenn ein generischer Mechanismus fachliche Semantik beansprucht, muss diese Behauptung an heterogenen realen Fällen geprüft werden. Kann der Core die behauptete Verantwortung nicht besitzen, ist Replacement ehrlicher als eine wachsende Optionsmatrix.

5.15 Kurzform für die Praxis

```
Wenn ein Test grün ist, frage:
  Welche Aussage hat er tatsächlich bewiesen?

Wenn ein Hook bequem ist, frage:
  Wer besitzt die Bedeutung dieses Hooks?

Wenn ein Core-State sichtbar ist, frage:
  Ist er autoritative Wahrheit oder nur eine Kopie?

Wenn ein Replacement droht, frage:
  Welche Mechanik bleibt unabhängig von der falschen Semantik wertvoll?
```

Das ist der Kern von „When Tested Architecture Must Be Replaced“: Nicht die Tests versagten. Der Architekturprozess wurde stark genug, um die Frage zu ändern, die getestet werden musste.

5.16 Reviewkarte: Getestete Architektur unter Semantikverdacht

Prüffrage	Warnsignal	mögliche Korrektur
Was bedeutet der grüne State?	nur intern gesetztes Label	realen Consumer-Output prüfen
Wer besitzt den Hook?	allgemeiner Core kennt Domainnamen	als normale Function verschieben
Was kann Rollback garantieren?	fremde Side Effects unbekannt	strukturelle und fachliche Garantie trennen
Hat ein leeres Artefakt Verhalten?	Core aktiviert es implizit	keine versteckte Semantik vergeben
Warum bleibt der Altpfad?	nur interne Gewöhnung	Replacement statt Compat-Flag
Welche Arbeit bleibt wertvoll?	„alles war falsch“	Mechanik und Evidenz separat inventarisieren

Der Review sollte erst schließen, wenn alter und neuer Contract gleichzeitig in einem Satz ausgedrückt werden können. Für diesen Fall:

> **Alt:** Der Core startet und stoppt jede Application entlang ihres Graphen.
> **Neu:** Der Core konstruiert und zerstört Graphen; ein Host ruft fachliche Aktivierungs- oder Cleanup-Functions explizit auf.

Diese Formulierung verhindert, dass das Replacement später wieder als bloßer Refactor gelesen und der alte Hook durch einen anderen Namen zurückgebracht wird.

6 DIX, zweiter Bogen: Als die „Bank“ wieder rausmusste

6.1 Eine Aktualisierung, die nicht in einen Nachsatz passt

Der erste Buchstand endete am 5. September 2026 bei DX-029. Damals war der universelle Application-Lifecycle gerade entfernt, die Typer-Composition aber noch nicht durch den vollständigen Application-E2E geschlossen. Fünf Tage später lag DIX bei einem technisch ganz anderen Stand: automatische Funktionsverträge waren gebaut, Norn und Strands waren als tragender Core abgenommen, ein spec-autoritatives Contract-System hatte sie weiter verschärft — und anschließend wurde diese gesamte verpflichtende Execution-Kette wieder aus dem stabilen Core entfernt.

Danach entstanden auf dem kleineren Core eine reale Multi-Application-CLI, ein lokaler modellierter State, eine optionale ROBA-Integration und eine ephemere Capability-Registry. Der letzte grüne Goal-Lauf wurde wiederum nicht einfach abgenommen: Ein unabhängiger Review fand eine Ticketgrenzverletzung und eine reale Rollback-Lücke. Beim Versuch, genau diesen Review zu belegen, produzierte der Evidence-Builder selbst erst einen ungültigen Run und danach einen versehentlich zu breiten Commit.

Das ist kein Nachtrag im Sinn von „inzwischen gibt es mehr Features“. Es ist eine zweite Fallstudie darüber, wie schnell eine technisch kohärente und ausdrücklich abgenommene Architektur ihre Gültigkeit verlieren kann — und wie ein Operationsprozess erst dann glaubwürdig wird, wenn er auch seine eigenen Fehler nicht wegredigiert. [E-034–E-046]

> **STATUSGRENZE** — Dieses Kapitel folgt dem fixierten Operations-Stand vom 10. September 2026. Mehrere Slices sind technisch umgesetzt; nicht jeder Block und nicht jedes Umsetzungspaket ist menschlich abgenommen. „Grün“, „technisch umgesetzt“ und „abgenommen“ bleiben verschiedene Aussagen.

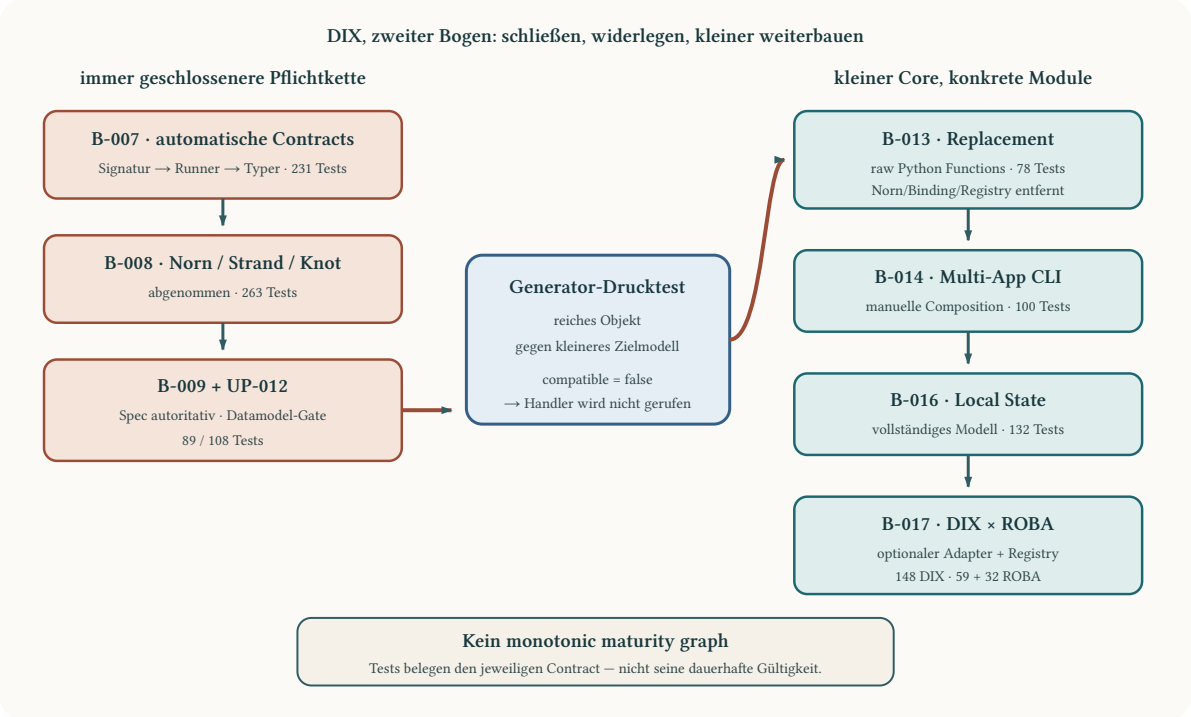

Abbildung 5: Der zweite DIX-Bogen: Eine immer geschlossenere Contract-Kette wird durch den Generator-Drucktest ersetzt; auf dem wieder kleineren Core entstehen konkrete Module.

6.2 B-007 — Der automatische Pfad funktioniert

Nach dem Lifecycle-Replacement lag eine naheliegende Integrationsschuld offen. Eine Application besaß bereits eine reale Python-Function-API. Die Typer-Schicht musste Parameter und Typen jedoch teilweise erneut beschreiben. B-007 schloss diese Lücke mit einer technisch eleganten Kette:

```
Python-Signatur
→ automatisch abgeleiteter FunctionContract
→ one-shot Application Runner
→ automatisch projizierte Typer-CLI
```

Der Contract blieb instanzlokal, der Runner erzeugte sein Target nur für einen Call und zerstörte dessen Graph im `finally`. Die Auto-CLI war ein normaler Consumer. Sie ersetzte weder die Application noch machte sie Typer-abhängig.

Der reproduzierte Lauf materialisierte ausschließlich Commit `831d5ed` und endete mit 231 grünen Tests. Help, gültige und ungültige Boolean-/Integer- Werte, Runner-Trace und Unload-Trace lagen gemeinsam vor. Technisch war das nicht bloß ein Entwurf: Der vollständige automatische Pfad funktionierte. [E-034]

Die verbleibende Frage war legitim. Ein aus Python abgeleiteter Vertrag hilft einem lokalen Adapter. Aber wie beschreibt man dieselbe fachliche Funktion, wenn ihre Implementierung später aus einer Datei, Datenbank, HTTP-Verbindung, ROBA-Runtime oder fremden Sprache kommen kann?

6.3 B-008 — Norn, Strand und Knot werden „eine Bank“

B-008 gab dieser Frage drei Namen:

```
Norn   = composition-scoped Registry, Binding und technische Ausführung
Strand = benannter Vertrag aus genau einem Input- und einem Output-Element
Knot   = Composition, die Strands implementiert oder kombiniert
```

Der Entwurf war bewusst nicht als Store, Workflow, Transport oder Service Registry gedacht. Ein Strand sollte nur sagen: Unter dieser ID nehme ich ein Element entgegen und liefere ein Element zurück. Ein gesamtes Datamodel konnte selbst als Elementgrenze auftreten. Cardinality, Query, Lazy Loading und Dependencies sollten nicht als parallele Flags in Norn wachsen, sondern durch konkrete Elemente und Funktionen komponiert werden.

Das war eine ernsthaft gute Reduktion. Sie entstand aus einer langen Folge verworfener Store-, Reader-/Writer-, Reference- und Capability-Flag-Modelle. Der zugehörige Lauf auf `7e8bb6c` belegte den Weg bis zur optionalen Typer-Projektion mit 263 grünen Tests. B-008 wurde am 7. September kooperativ abgenommen; im Operations-Dokument steht die menschliche Bewertung, der Schnitt sei „eine Bank“. [E-035]

> **WERKSTATT · DER GEFÄHRLICHE MOMENT** B-008/B-013 · OPERATIONS
>
> Das war keine ironische Fehlentscheidung. Genau deshalb ist der Fall wichtig. Wir hatten nicht irgendeinen halbgaren Entwurf euphorisch überschätzt. Wir hatten eine kleine Sprache, einen realen E2E, lokale Scopes und eine saubere Trennung von Vertrag, Binding, Implementierung und Projektion. Das Ding trug den Druck, den wir ihm gegeben hatten. Nur der nächste Druck war härter.

Wer diesen Moment später nur mit „Norn war falsch“ zusammenfasst, zerstört die eigentliche Evidenz. Falsch war nicht, dass funktionale Datenverträge nützlich sein können. Falsch wurde die Folgerung, dass **jede** exponierte lokale Function durch dieselbe Contract- und Execution-Schicht laufen müsse.

6.4 B-009 und UP-012 — Die falsche Grenze wird immer sauberer

Der nächste Schritt reagierte erneut auf ein reales Problem. Wenn der Contract automatisch aus Python entsteht, verändert eine Codeänderung möglicherweise unbemerkt die exponierte Schnittstelle. B-009 ersetzte deshalb die Autoritätsrichtung:

```
contract.toml   = autoritative Sollform
Python-Code     = Implementierung
FunctionBinding = geprüfte Verbindung
```

Keine automatische Contract-Erzeugung, keine stille Spec-Mutation, kein `fallback_any`: Aus Sicht eines sprachübergreifenden Vertrags ist diese Strenge attraktiv. Das System erhielt Contract-only-Module, exakte Versionen, atomare Publication und harte Binding-Fehler. Auf Commit `e300d47` waren 89 Tests grün. Der Stand war technisch umgesetzt, aber nicht menschlich abgenommen. [E-036]
UP-012 schloss anschließend die noch sichtbare Lücke für strukturierte Daten: codefreie Models im Modul, Model-Referenzen aus Contracts, lokale Norn-/Datamodel-Bindings, Input- und Output-Verarbeitung. Auf `4bde81c` liefen 108 Tests. Der Run dokumentierte ausdrücklich, dass kein Parallelpfad mehr bestand. [E-037]
Die Kette sah nun beeindruckend vollständig aus:

```
Model Registry
→ Contract ModelReference
→ lokales Norn-Binding
→ Datamodel- und Element-Verarbeitung
→ Python-Function
→ Output-Verarbeitung
```

Jeder einzelne Schritt machte den vorherigen Entwurf konsistenter. Genau das ist eine der unangenehmsten Architekturfallen: Lokale technische Qualität kann die falsche Ownership nicht nur verstecken, sondern befestigen. Je sauberer Parser, Registry, Binding und Fehlermodell werden, desto teurer fühlt sich die Frage an, ob diese Kette überhaupt an dieser Stelle existieren darf.

6.5 Der Generator-Drucktest trifft nicht den Generator

B-012 sollte den nächsten Ausbau vorbereiten: einen portablen Generator, der mit codefreien Model-/Strand-Verträgen arbeitet. Der entscheidende Test war unspektakulär. Ein reiches Objekt und eine kleinere Sicht sollten unabhängig bleiben:

```
user      = {id, age, meta}
user_meta = {nickname, description}

user gegen Modell {id}
→ id ist bekannt
→ age und meta sind zusätzliche Felder
→ der konkrete Consumer entscheidet, was das bedeutet
```

Die ursprüngliche Datamodel-Semantik war beobachtend. Sie konnte bekannte Werte sowie Missing-, Additional- und Type-Issues liefern, ohne daraus eine globale Zulassungsentscheidung zu machen. Im verpflichtenden Norn-Pfad galt inzwischen jedoch effektiv:

```
DatamodelResult.compatible == false
→ ElementResult.value = None
→ Handler wird nicht aufgerufen
```

Damit war Datamodel unbemerkt von einer optionalen technischen Capability zur allgemeinen Admission Policy jedes Function-Calls geworden. Der Generator war nicht das neue Feature, das diesen Fehler erzeugte. Er war das erste Gegenbeispiel, das ihn sichtbar genug machte. [E-038]
Naheliegende Reparaturen gab es sofort:

- `strict = false`;
- `project` statt `validate`;
- `report_only`;
- unbekannte Felder tolerieren;
- `Any` als Fallback;
- generator-spezifische Strandregeln.

Jede Variante hätte den Test vermutlich grün gemacht. Keine beantwortete, warum der Core für Generator, Validator, Sicherheitsgrenze und Remote-Call dieselbe Policy besitzen darf. Sobald konkurrierende fachliche Bedeutungen als Modi einer vermeintlich universellen Engine erscheinen, ist eine größere Optionsmatrix oft kein Flexibilitätsgewinn, sondern ein Ownership-Alarm.

6.6 B-013 — Replacement statt Reparaturmodus

B-013 zog den harten Schnitt:

```
entfernen:
  contract.toml als Pflicht im stabilen Core
  Norn als unumgehbare Execution-Schicht
  runtimeweite Model-Registry
  FunctionBinding als Pflicht

behalten:
  explizites Module Load/Unload
  isolierte Composition-/Application-Graphen
  lokale Dependencies
  explizite Function-Exposition
  Element und Datamodel als optional angeforderte Capabilities
  reale Python-Signatur als Autorität des lokalen Calls
```

Specs blieben autoritativ — aber nur für das, was sie tatsächlich besitzen: Graph Assembly und Exposition. Sie bestimmten nicht länger automatisch die Input-/Output-Policy jedes lokalen Calls. Functions erhielten und lieferten wieder rohe Python-Werte. Ein sprachübergreifender Contract blieb als späteres Modul oder Grenzartefakt möglich; er war nur keine Pflichtschicht mehr.

UP-013 setzte das Replacement in drei Commits um. Der Lauf belegte auf `de00b83` 78 Tests, Source- und Wheel-Seed, beliebige Python-Objekte, echte Signaturen, Instanzisolation sowie Load-/Destroy-/Unload-Grenzen. Das Wheel enthielt weder die entfernten Contract-/Norn-Core-Pfade noch `unstable`. B-013 blieb menschlich offen. [E-039]

> **ANAM · SPÄTERE EINORDNUNG** B-013 · INTERPRETATION
>
> Die wichtige Korrektur lautet nicht „spec-first war falsch“. Das wäre nur die nächste pauschale Parole. Eine Spec kann für Assembly autoritativ, für einen Remote-Wire-Contract zwingend und für einen lokalen Python-Call trotzdem die falsche Laufzeitpolizei sein. Autorität braucht immer ein Objekt und eine Grenze: autoritativ wofür?

6.7 Warum 263 Tests nicht mehr sind als 78

Die Zahlen dieses Bogens fallen und steigen:

Stand	technischer Lauf	Ergebnis
B-007 Auto-Contracts	`831d5ed`	231 passed
B-008 Norn/Strands	`7e8bb6c`	263 passed
B-009 Spec-Contracts	`e300d47`	89 passed
UP-012 Model/Norn	`4bde81c`	108 passed
UP-013 Replacement	`de00b83`	78 passed
UP-014 Multi-App-CLI	`2a07c6a`	100 passed
UP-017 lokaler State	`46f3e0c`	132 passed
UP-018 DIX–ROBA	`b03b447`	141 passed
UP-019 Capability-Registry	`f11a60b`	147 passed
RV-001 Korrektur	`9f14223`	148 passed

Diese Tabelle ist **keine** Fortschrittskurve. B-009 hatte weniger Tests, weil große historische Flächen nach `unstable` verschoben oder entfernt wurden. UP-013 hatte weniger Tests, weil es eine verpflichtende Architektur samt Testsuite ersetzte. Die Zahlen sind nur innerhalb des jeweiligen Claims und Source-Stands aussagekräftig. Ein monoton wachsender Testcounter hätte hier sogar falsche Stabilität belohnen können. [E-050]

Die bessere Frage lautet:

> Welche Semantik umfasst diese Suite, und welche entfernte Semantik soll sie gerade **nicht** mehr konservieren?

6.8 B-014 — Nach dem Rückbau zuerst wieder etwas benutzen

Der erste Aufbau auf dem bereinigten Core war bewusst konkret. B-014 nahm zwei unabhängige Applications, eine spezialisierte Typer-Composition und eine kleine manuelle Assembly-Application. Daraus entstand per bestehendem Bootstrap ein ausführbares Multi-Command-CLI:

```
my_cli base render ...
my_cli test hello ...
my_cli test hello_world ...
```

Typer blieb optional. Function-, Group- und Parameter-IDs wurden exakt erhalten. Parameter waren named-only; reale Python-Annotationen blieben die lokale Typquelle. Auswahl, Rename oder Adaption geschah durch normale Wrapper, nicht durch eine zweite CLI-Policy-Sprache.
Der E2E fand einen realen Bootstrap-Bug: Der erste Multi-Modul-Launcher enthielt durch zwei Separatoren `),,`. Der bisherige Single-Modul-Seed hatte das nicht sichtbar gemacht. Die Reparatur blieb außerhalb des Core. Source und Wheel liefen anschließend mit 100 grünen Tests; `src/dix/core` blieb bytegenau unverändert. [E-040]
Das ist die konstruktive Hälfte eines Replacements: Nach dem Entfernen der falschen Universalität muss der kleinere Kern wieder eine echte Arbeit tragen. Sonst ist „weniger Core“ nur ästhetische Behauptung.

6.9 B-015 — Der Prozess entdeckt seinen eigenen Drift

Während der Core sauberer wurde, war der Operationsprozess selbst still schwächer geworden. B-015 verglich die Artefakte statt nur ihre Namen:

```
Block existiert
=/Herleitung ist append-only und nachvollziehbar

testlaeufe/run-001 existiert
=/Lauf wurde ausgeführt und ist reproduzierbar
```

B-008 war der letzte Block mit konsistent fortgeschriebener Herleitung. B-009 war der letzte Run mit Builder, fixiertem Source-Snapshot, Rohoutputs und Provenienz; spätere Run-Verzeichnisse waren teilweise nur redaktionelle Berichte. Der sichtbare Prozess hatte seine Semantik behalten wollen und nur seine Ordnernamen behalten. [E-041]
B-015 führte deshalb einen Vorwärts-Cutover ein:

- Root-`AGENTS.md` als Router;
- getrennte Instruktionen für Block, Umsetzungspaket, Tickets und Testlauf;
- Block mit aktuellem Zielstand **und** fortlaufender Herleitung;
- explizite Slice-Freigabe vor dem UP;
- committed UP vor committed Tickets;
- Builder plus unveränderlicher `run-NNN` mit Source, Flow, Commands, Rohoutputs und Exitcodes;
- menschliche Abnahme als eigener Status;
- kein historischer Backfill.

Der letzte Punkt ist entscheidend. Die schwächeren alten Runs wurden nicht nachträglich in einen Prozessstandard umgeschrieben, den es damals nicht gab. Der Drift selbst blieb Evidenz. B-015 wurde abgenommen.

6.10 B-016 — Config wird kleiner, indem sie zu State wird

Die nächste fachliche Frage begann als Config. Ein erster Slice, UP-016, erzeugte aus einer owner-controlled deklarativen Mapping-Spec einen echten Pydantic-Modelltyp. Pydantic blieb optionales First-party-Modul und nicht DIX-Core. Der Resolver nahm keine Values, Sources, Merge-Regeln oder Persistence an. Auf `b2d5d14` liefen 118 Tests.
Der Druck ging anschließend nicht in Richtung eines größeren Config-Systems. Die wiederverwendbare Mechanik war allgemeiner und gleichzeitig kleiner:

```
dix/state/models
  deklarative StateModelSpec → type[BaseModel]

dix/state/local
  eine owner-relative Model-Datei
  + optionaler Initialwert
  → get() / set(complete_mapping)
```

UP-017 ersetzte `dix/config` ohne Kompatibilitätsfassade durch `dix/state`. `get()` liefert einen abgetrennten Dump. `set()` validiert einen vollständigen Kandidaten, übernimmt ihn atomar innerhalb dieser lokalen Instanz und meldet nur, ob sich der normalisierte Wert geändert hat. Bei ValidationError bleibt der alte Wert erhalten. Absichtlich fehlen:

- Actions, Events und Callbacks;
- Teilupdates und Pfadsyntax;
- Sources, Merge und Prioritäten;
- Registry, Caches und Prozess-Sharing;
- File-Persistenz;
- Config-, Typer-, Exec- oder ROBA-Sondersemantik.

Ein Owner kann `get` und `set` transparent exportieren oder mit normalem Composition-Code wrappen und dort auf `changed` reagieren. Mehrere State-Instanzen entstehen durch normale Composition-Aliases, nicht durch eine globale State-Registry. Der Run auf `46f3e0c` belegte 32 Zieltests, 132 Tests gesamt, Source und Wheel sowie die Abwesenheit der alten Config-Oberfläche. UP-017 und danach B-016 wurden menschlich abgenommen. [E-042] Diese Entscheidung ist kein Widerspruch zur Kritik an universellem State. DIX definiert hier weder „Workspace“ noch „Editor“ noch eine gemeinsame fachliche Wahrheit. Es bietet einem konkreten Owner eine kleine lokale, modellvalidierte Wertgrenze. Ob daraus Config, CLI-State oder etwas anderes wird, entscheidet der aufbauende Consumer.

6.11 Der Status am Ende des zweiten Bogens

Block / Slice	fachlicher Stand am Snapshot
B-007	umgesetzt; Operations-Artefakte weiterhin untracked, separat fixiert
B-008	menschlich abgenommen; für den Core später durch B-013 ersetzt
B-009	technisch umgesetzt; keine menschliche Abnahme
B-010	abgestimmt, aber nicht als eigener Zielstand umgesetzt
B-011	abgenommen; minimaler Seed bleibt erhalten
B-012 / UP-012	Block offen; technischer Drucktest grün, architektonisch nicht abgenommen
B-013 / UP-013	Replacement technisch umgesetzt; menschliche Blockabnahme offen
B-014 / UP-014	technisch umgesetzt; fachliche Abnahme offen
B-015	Operations-Cutover menschlich abgenommen
B-016 / UP-017	lokaler modellierter State menschlich abgenommen
B-017 / UP-018 / UP-019	Integration technisch weit fortgeschritten; menschliche Gesamt-/Slice-Abnahmen offen
RV-001	technisch umgesetzt; menschliche Abnahme offen

Diese Gleichzeitigkeit ist kein Verwaltungsfehler. Ein früher abgenommener Block kann als historische Entscheidung gültig gewesen und für den heutigen Core ersetzt sein. Ein später technisch grüner Slice kann noch auf menschliche Abnahme warten. Ein offener Gesamtblock kann bereits explizit freigegebene und umgesetzte Teilslices enthalten. Status braucht deshalb immer ein Objekt und einen Zeitpunkt. [E-048, E-049]

6.12 Was dieser Bogen dem ersten hinzufügt

Der Lifecycle-Fall zeigte: Grüne Tests können eine falsche universelle Semantik präzise beweisen. Der Norn-Fall verschärft die Aussage:

1. Die Architektur kann klein, elegant, lokal isoliert und menschlich abgenommen sein.
2. Ein nachfolgender Ausbau kann sie technisch immer konsistenter machen.
3. Erst ein anderes Gegenbeispiel kann zeigen, dass die gemeinsame Pflichtschicht selbst die falsche Eigentumsbehauptung war.
4. Lokale Reparaturmodi können den Fehler kaschieren, statt ihn zu lösen.
5. Replacement darf die wertvollen Teile — Verträge an echten Grenzen, Datamodel als Capability, isolierte Graphen — getrennt bewahren.
6. Der erste Aufbau danach muss praktisch beweisen, dass der kleinere Kern noch trägt.

Und B-015 fügt eine zweite Ebene hinzu: Dieselben Regeln gelten für den Entwicklungsprozess. Ein Builder, ein Ticket oder eine `AGENTS.md` ist keine magische Wahrheitsmaschine. Auch Evidence-Werkzeuge brauchen Scope, Fehlerpfade, Review und sichtbare Korrekturhistorie.

Der nächste Abschnitt über DIX und ROBA ist deshalb keine futuristische Integrationsskizze mehr. Er zeigt, was geschah, als der wieder verkleinerte DIX-Core auf eine reale fremde Public API, einmalige Tokens, lokale Capability-Sockets und prozessübergreifenden Rebind traf.

7 Übertragbare Architekturprinzipien

7.1 Prinzipien sind keine universellen Gesetze

Die folgenden Prinzipien wurden aus DIX und ROBA abstrahiert. Sie sind Entscheidungsheuristiken mit Gegenpositionen, keine beweisbaren Gesetze. Jedes Prinzip nennt deshalb seine Geltungsbedingung und den typischen Missbrauch.

7.2 1. Behavior-Contract vor universellem State-Modell

Wenn heterogene Systeme kooperieren sollen, beginnt die Diskussion oft mit gemeinsamen Daten: Buffer, Window, Workspace, Task, Session. Dadurch wird die Ontologie des ersten Systems zur Norm.
Ein Function-Contract beginnt enger:

```
open(target)
render(value)
create(input)
describe() -> descriptor
```

Jeder Provider besitzt seinen nativen State und implementiert nur die Fähigkeit, die er wirklich garantieren kann. Ein gemeinsamer State kommt erst dann hinzu, wenn er selbst eine fachliche Source of Truth ist.
Gilt besonders, wenn Tools unterschiedliche interne Modelle besitzen.
Gegenposition: Für echte kollaborative Domänen — etwa gemeinsames Dokument, Order oder Lease — ist autoritativer Shared State zentral. Dann wäre Function-only eine Vermeidung notwendiger Konsistenzsemantik.
Missbrauch: Jede Datenabfrage als Function verstecken, obwohl mehrere Consumer dieselbe versionierte Datenstruktur brauchen.

7.3 2. Native Capability vor Compatibility Interface

bwrap, Docker und Windows-Prozessisolation unterscheiden sich in Namespace, Mount-, Credential- und Lifecycle-Semantik. Eine frühe gemeinsame `Sandbox.start()`-Schnittstelle hat zwei schlechte Optionen:

- nur den kleinsten gemeinsamen Nenner anbieten;
- target-spezifische Optionen durch generische Flags schleusen.

Native Module dürfen ihre volle Semantik besitzen. Wenn ein konkreter Consumer einen gemeinsamen Teil benötigt, entsteht ein separates Compatibility-Artefakt:

```
bwrap module ─┐
docker module ├─ optional isolation.compat profile
windows module┘
```

Gilt besonders, wenn Security- oder Fehlergarantien targetabhängig sind.
Gegenposition: Für eine eng definierte Organisation kann ein bewusst begrenztes gemeinsames Interface erhebliche Bedien- und Testvorteile bieten.
Missbrauch: „Nativ“ als Ausrede verwenden, um jede Integration oder gemeinsame Terminologie zu vermeiden.

7.4 3. Mechanismus und Policy dürfen getrennt wachsen

ROBA Core kann Contexts, State, Instances und ACL-Principals verwalten. Er weiß nicht, was „privates Profil“, „Firmenprofil“, „Workspace“ oder „Architecture Mode“ bedeutet. Diese Trennung ist keine Abwertung persönlicher Policy. Sie erlaubt gerade, diese Policy als starkes eigenes Artefakt zu entwickeln.
Mechanismus beantwortet: Was kann zuverlässig getan werden?
Policy beantwortet: Wann, für wen und mit welcher Bedeutung wird es getan?

Gegenposition: Manche Policy ist systemische Sicherheitsinvariante und darf nicht optional im Userlayer liegen. Zugriffskontrolle selbst ist Mechanismus; die Mindestanforderung „Production Secrets nie in untrusted Sandboxes“ kann verbindliche Organisationpolicy sein.

7.5 4. Core versus Module ist eine Ownership-Frage

„Core soll klein sein“ ist zu oberflächlich. Eine Capability gehört in den Core, wenn:

- jede darüberliegende Schicht auf exakt dieselbe Bedeutung angewiesen ist;
- der Core die Garantie vollständig kontrollieren kann;
- ein externer Aufbau nur denselben Mechanismus duplizieren würde;
- die Semantik nicht an einen Zielhost oder Workflow gebunden ist.

Sie gehört eher in ein Module, wenn:

- unterschiedliche Ziele legitime inkompatible Bedeutungen haben;
- sie eine optionale Technologieabhängigkeit trägt;
- sie fachliches Behavior statt strukturelle Mechanik ist;
- mehrere alternative Policies möglich sind.

DIX Core besitzt strukturelle Graphlebenszeit. Das Typer-Binding ist Module. ROBA Core besitzt ACL- und Stateprimitive. Hub-Recovery ist externer Consumer.

7.6 5. Autorität gilt pro Grenze, nicht global

Eine Function-Signatur in Python und eine Kopie in TOML erzeugen zwei Quellen:

```toml
[[function]]
name = "render"
parameters = [{name="value", type="string"}]
```

```python
def render(self, value: str, *, strict: bool = False) -> str: ...
```

Schon ein neuer Keywordparameter erzeugt Drift. Für lokale in-process Functions kann der Code deshalb die Signaturautorität sein; deklarative Specs beschreiben dann Auswahl, Alias, Dependencies und Exposition, nicht eine zweite Behavior-Sprache.

Der zweite DIX-Bogen zeigt jedoch die notwendige Schärfung. B-009 machte einen deklarativen Contract zur allgemeinen Autorität und ließ ihn durch Datamodel- Kompatibilität über die Zulässigkeit konkreter Werte entscheiden. Das war für die lokale Generatorgrenze zu stark: Ein reiches, gültiges Objekt wurde verworfen, nur weil es nicht verlustfrei in ein kleineres Beobachtungsmodell passte. B-013 entfernte dieses globale Admission-Gate wieder. [E-036–E-039]

Die Regel lautet daher nicht „Code gewinnt immer“, sondern:

> **INVARIANTE** — Genau die Grenze, deren Stabilität gegenüber ihren Consumern garantiert werden muss, besitzt den autoritativen Contract. Eine Beobachtungs- oder Projektionsfähigkeit darf nicht nebenbei zur globalen Zulassungspolitik werden.

Gegenposition: In sprachübergreifenden oder Remote-Protokollen muss ein maschinenlesbarer IDL-Contract autoritativ sein. Dann wird Code generiert oder gegen die IDL validiert.

Missbrauch: „Code ist Wahrheit“ behaupten, obwohl externe Consumer den Code nicht inspizieren können und stabile Versionierung brauchen.

7.7 6. Deklarative Specs ohne Behavior-DSL

Deklaration ist stark für Struktur:

- welche Dependencies werden verwendet?
- welche Function wird exponiert?
- welche Configwerte gelten?
- welche Module Roots sind vertrauenswürdig?

Sobald TOML Schleifen, Bedingungen, Fehlerbehandlung und Nebenwirkungsreihenfolge ausdrückt, entsteht eine zweite Programmiersprache. DIX hält Behavior in normalem Python-Code und Specs als inspectable Contract.

Gegenposition: Eine begrenzte DSL kann in hochstandardisierten Domänen sicherer, analysierbarer und für Nichtentwickler zugänglich sein.
Prüffrage: Ist die DSL-Semantik selbst das Produkt — oder verstecken wir nur Code in YAML/TOML?

7.8 7. Explizite Wrapper markieren Ownership

Wrapper sind sinnvoll, wenn ein Consumer eine fremde Function bewusst in seine eigene API übernimmt. Sie bilden:

- lokalen Namen;
- lokale Signatur;
- Adaptionspunkt;
- Fehlerübersetzung;
- Deprecation-/Migrationsgrenze;
- inspectable Herkunft.

Ein Generator kann Wrappergerüste erzeugen, aber die lokale Entscheidung nicht ersetzen. Automatisch alles zu re-exportieren würde aus Dependency wieder implizite API machen.

7.9 8. Lokale Dependency-Graphen und Instanzisolation

Globale Registries sind bequem, bis parallele Konfiguration, Tests oder Mandanten entstehen. DIX unterscheidet:

- Definition: statischer Contract;
- Instanz: konkreter lokaler Graph;
- Owner-Scope: Lebenszeit- und Sichtbarkeitsgrenze.

Diese Trennung ermöglicht zwei Instanzen derselben Definition ohne Statekollision. Shared Capabilities bleiben möglich, müssen aber explizit als runtime-scoped gesetzt werden.
Gegenposition: Manche Ressourcen sollen Singleton sein. Dann sollte der Singleton-Contract selbst explizit sein, nicht zufällige Folge einer globalen Registry.

7.10 9. Autoritativer State statt synchronisierter Kopien

Wenn Neovim, Robac und ein Webclient denselben Contextzustand brauchen, gibt es zwei Modelle:

1. Jeder hält eine Kopie und synchronisiert Events.
2. Ein Dienst besitzt den autoritativen State; Clients lesen und schreiben über seine API.

ROBA implementiert Modell 2 für bewusst dort gehaltenen Context-/Instance-State. Das reduziert Konfliktquellen, löst aber nicht automatisch Offlinebetrieb, Caching, Verfügbarkeit oder Persistenz. Im aktuellen ROBA-Contract ist dieser State ephemer: Seine Autorität gilt innerhalb der laufenden Runtime, nicht über Daemon- oder Maschinenverlust hinweg.
Invariante: Ein DIX-ROBA-Modul darf State nicht still in DIX spiegeln. Ein Output kann Snapshot/Descriptor sein, aber die Authority bleibt ROBA.

7.11 10. Drei Autorisierungsebenen bleiben getrennt

In der inzwischen technisch umgesetzten DIX-ROBA-Integration können drei Prüfungen gelten:

1. **DIX Session:** Darf dieser Teilnehmer die konkrete Interfacefläche nutzen?
2. **Function Exposition:** Ist die Function in dieser Application überhaupt veröffentlicht?
3. **ROBA ACL:** Darf der verwendete Principal die Zielresource lesen, schreiben, managen oder Grants vergeben?

Diese Ebenen sind nicht redundant:

```
Session erlaubt Interface
  =/Function ist exponiert
  =/Backend-Principal besitzt Resource-Recht
```

Eine allumfassende „authenticated“-Boolean würde Fehlerdiagnose und Least-Privilege zerstören.

7.12 11. Trust Boundary ist nicht Contract

DIX lädt Python-Code nur aus trusted Module Roots. Das ist eine Admission- Entscheidung. `composition.toml` beschreibt anschließend Struktur und publizierte Functions. Es sandboxed den Code nicht.

ROBA validiert Inputs sowohl im Client als auch erneut am Daemon. Die zweite Validierung ist Trust Boundary; die erste verbessert UX und verhindert unnötigen Transport.
Prüffrage: Welche Annahme schützt vor bösartigem oder fehlerhaftem Code, und welche beschreibt nur erwartete Kooperation?

7.13 12. Struktureller Lifecycle versus fachliches Verhalten

Der B-005/B-006-Fall liefert die klare Formel:

- Core darf eigene Definitionen, Instanzen, Referenzen und Graphen verwalten.
- Domain darf Aktivierung, Readiness, Retry, Shutdown und externe Ressourcen definieren.

Ein spezialisiertes Service-Host-Modul kann ein reiches Lifecycle-Protokoll anbieten. Der allgemeine Graph-Core darf es nicht implizit ausführen.

7.14 13. Standards als ausführbare Artefakte

Ein Standard wie „CLI-Eingaben sind named-only und Env-Namen explizit“ kann auf drei Arten existieren:

1. als Erinnerung im Prompt;
2. als Textkonvention im Handbook;
3. als generator-, validator- und testbares CLI-Modul.

Die dritte Form ist am belastbarsten. Text bleibt nötig, um die Begründung zu erklären. Prompt bleibt nützlich, um einen Lauf zu steuern. Aber das ausführbare Artefakt macht Abweichung objektiv sichtbar.
Gegenposition: Nicht jede Regel rechtfertigt Tooling. Kleine, seltene oder noch explorative Regeln sollten nicht vorschnell in Generatoren gegossen werden.

7.15 14. Ehrliche Unsupported-Semantik

Portabilitätslayer neigen dazu, fehlende Fähigkeiten mit leeren Werten, No-ops oder Best-effort zu kaschieren. Das erzeugt nominelle Kompatibilität, aber unzuverlässige Bedeutung.
Ehrlicher ist:

```
supported with contract X
unsupported
requires adapter artifact Y
```

Ein Editor ohne `list_open_files` darf diese Capability nicht anbieten. Ein Windows-Isolationsmodul muss keinen Unix-Socket simulieren. Ein Compatibility-Artefakt kann bewusst einen kleineren Contract definieren und für Unsupported hart fehlschlagen.

7.16 15. Replacement vor Optionsmatrix

Wenn eine Grundbedeutung falsch ist, wirkt ein Kompatibilitätsflag freundlich:

```
legacy_lifecycle = true
strict_mode = false
compat_workspace = host
```

Jedes Flag konserviert jedoch einen zweiten Architekturpfad. Vor öffentlicher Stabilisierung ist ein klares Replacement oft billiger und verständlicher.
Gegenposition: Bei produktiven externen Nutzern kann Migration wichtiger sein als interne Reinheit. Dann braucht es versionierten Altvertrag, Deprecationpfad und messbaren Abschaltpunkt — nicht stillen Bruch.

7.17 16. Negative Evidenz ist ein Ergebnis

Ein Drucktest kann zeigen, dass:

- eine Capability an der falschen Schicht liegt;
- eine Ontologie nur einen Consumer beschreibt;
- ein Rollbackversprechen externe Side Effects nicht kontrollieren kann;
- eine Compatibility-Schnittstelle Securitysemantik verschleiert;
- ein vermeintlich generisches Feature gar keinen zweiten realen Consumer hat.

Solche Ergebnisse müssen in der Evidence Chain bleiben. Wer nur bestätigte Entscheidungen dokumentiert, produziert Architekturtheater.

7.18 17. Komposition ersetzt keine Verantwortung

DIX macht es leicht, Functions und Graphen zu kombinieren. Das legitimiert nicht jede Kombination. Ein Application-Contract muss weiterhin erklären:

- wer das Ergebnis besitzt;
- welche Side Effects auftreten;
- welcher Principal verwendet wird;
- wie Fehler sichtbar sind;
- was wiederholter Aufruf bedeutet;
- wer Destruction und externes Cleanup verantwortet.

Komposition ist Mechanismus. Verantwortung bleibt Architekturarbeit.

7.19 18. Optionaler lokaler State ist keine globale Ontologie

Nach der Kritik an universellem Shared State wäre die einfache Reaktion, State komplett aus DIX zu verbannen. B-016 zeigt die präzisere Grenze:

```
deklariertes Pydantic-Modell
→ genau eine composition-lokale Modellinstanz
→ get() als vollständiges Mapping
→ set(full_mapping) als validierendes Replacement
```

Die Capability definiert weder Workspace noch Editor noch gemeinsame Persistenz. Sie gibt einem konkreten Owner nur eine modellierte lokale Wertgrenze. Zwei Composition-Instanzen bleiben isoliert. Wer Persistenz, partielle Updates, Events oder verteilte Konsistenz braucht, muss diese Semantik als eigenes Artefakt hinzufügen. [E-042]

Gegenposition: Ein Full-model-Replacement kann bei großen oder konkurrierenden Modellen ineffizient und konfliktanfällig sein. Dann ist ein anderer State-Contract nötig — nicht ein schleichend wachsendes `set()`.

7.20 19. Grüne Funktion ist nicht grüne Architekturgrenze

Eine Suite kann beweisen, dass das Endsystem tut, was ihr Testmodell verlangt, und trotzdem verschweigen, dass:

- Ticket A bereits die Verantwortung von Ticket B enthält;
- ein Commit nicht mehr isoliert reviewbar ist;
- ein Cleanup nur den eindeutig fehlgeschlagenen Fall modelliert;
- ein Operations-Run aus der falschen Sourcefläche gebaut wurde.

Darum braucht ein riskanter Slice nach dem Featurelauf einen Review, dessen Fragen nicht einfach aus denselben Akzeptanzkriterien kopiert sind. RV-001 fand genau auf dieser Ebene Fehler, obwohl alle Featuretests grün waren. [E-045]

7.21 20. „Create fehlgeschlagen“ ist kein eindeutiger Zustand

Bei externen Mutationen gibt es mindestens drei relevante Ergebnisse:

```
Ressource nicht erzeugt
Ressource erzeugt und Erfolg bestätigt
Ressource erzeugt, aber Bestätigung verloren
```

Der dritte Fall entsteht bei Timeouts, abgebrochenem Transport oder verlorener Antwort. Rollbackcode darf Cleanup deshalb nicht ausschließlich von einem erhaltenen Returnwert abhängig machen. Er muss den Versuch vor der Mutation markieren, eine idempotente oder sicher klassifizierbare Kompensation versuchen und nur nachweislich harmlose Fehler unterdrücken.

Gegenposition: Nicht jede externe API erlaubt sicheres Retry oder Idempotenz. Dann muss der Zustand ausdrücklich als unbekannt eskaliert werden; ein lokales `false` wäre erfundene Gewissheit.

7.22 21. Evidence ist selbst fehlbar und deshalb append-only

Ein Artifact Builder kann Shellcode falsch quoten. Ein angeblicher Snapshot kann eine vom Test erwartete Git-Provenienz verlieren. Ein Staging-Check kann ohne fail-fast-Kette folgenlos scheitern. Darum gelten für Evidence dieselben Grundregeln wie für Produktcode:

- exakte Inputs und Source-Commits;
- explizite Exitcodes;
- Prüfsummen und Rohoutput;
- negative Pfade;
- kein stilles Überschreiben eines fehlgeschlagenen Runs;
- sichtbare Korrektur durch einen neuen Run oder Commit.

Ein ungültiger Run belegt den fachlichen Claim nicht. Seine erhaltene Fehlerursache belegt aber, dass der Prozess seine eigene Fehlbarkeit nicht aus der Geschichte löscht. [E-046, E-047]

7.23 Eine kompakte Entscheidungsmatrix

Frage	Eher Core	Eher Module/Consumer	Alarmzeichen
Wer kontrolliert die Garantie?	Core vollständig	Zielsystem/Host	niemand vollständig
Wie viele legitime Semantiken?	eine	mehrere	Optionsmatrix wächst
Ist State autoritativ?	ja, Core-domain	externer Dienst/Tool	mehrere synchronisierte Kopien
Ist Behavior deklarativ?	strukturell	normaler Code	DSL wächst unbemerkt
Ist Trust geklärt?	enge Boundary	natives Modul	Config wird als Sandbox missverstanden
Gibt es realen zweiten Consumer?	nachgewiesen	noch spezifisch halten	hypothetische Universalität
Was passiert bei Unsupported?	harter Contractfehler	nativer Fehler/Adapter	No-op oder leerer Fakewert
Was bedeutet „autoritativ“?	innerhalb benannter Runtime	externe dauerhafte Quelle	Ephemeralität bleibt ungesagt
Ist ein Create eindeutig fehlgeschlagen?	serverseitig belegt	Zustand als unbekannt eskalieren	Cleanup nur nach Returnwert
Wer prüft den Beleg?	unabhängiger Reviewpfad	benannte manuelle Abnahme	Builder erklärt sich selbst für gültig

Diese Matrix entscheidet nicht automatisch. Sie zwingt nur die verborgene Begründung an die Oberfläche.

8 DIX × ROBA: Vom Zielbild zum optionalen Vertikalschnitt

8.1 Status dieses Kapitels

> **VERIFIED / TECHNISCH UMGESETZT / MENSCHLICH OFFEN** — Die in V1 noch hypothetische Abhängigkeitsrichtung ist inzwischen implementiert: Das optionale First-party-Modul `dix/roba` konsumiert ausschließlich die öffentliche Python-API des eigenständigen Pakets `roba`. Source-, Wheel- und Cross-Process-Läufe sind grün. Die fachliche Gesamtannahme von B-017 sowie UP-018, UP-019 und RV-001 waren am Snapshot noch nicht menschlich abgenommen. [E-043–E-048]

Das ist eine absichtlich sperrige Statuszeile. Sie verhindert drei bequeme, aber falsche Kurzschlüsse:

1. Die Integration ist nicht mehr bloß ein Zielbild.
2. Technische Existenz ist noch keine fachliche Abnahme.
3. Ein grüner E2E macht ein offenes Review nicht rückwirkend bedeutungslos.

8.2 Was V1 erwartet hatte — und was tatsächlich zuerst gebaut wurde

Die vorige Ausgabe schlug als kleinsten Integrationszyklus vor, aus einer DIX-Application genau einen ROBA-Context-Wert zu lesen und einen Instance-Wert mit minimaler ACL zu schreiben. Das war ein plausibler Drucktest, aber nicht der später gewählte erste Schnitt.

Die Operationsarbeit setzte vorher an. Bevor DIX fachlichen State aus ROBA konsumiert, muss es eine fremde Runtime überhaupt ohne versteckte Hostpolicy konfigurieren, starten, adressieren und wieder stoppen können. UP-018 baute daher diese Kette:

```
DIX Local State
→ explizite ROBA-Konfiguration
→ öffentliche ROBA-Python-API
→ Daemon start / status / stop
→ DIX Application
→ manuell komponierte Typer-CLI
```

Das ist keine Niederlage der alten Hypothese. Es ist ihr erster Kontakt mit dem realen Baupfad. Der kleinere Druck lag nicht bei einem beliebigen Statewert, sondern an der Runtime- und Konfigurationsgrenze. [E-043]

8.3 Zwei Systeme bleiben zwei Systeme

Die Integration trägt, weil sie keine nachträgliche Verschmelzung versucht:

```
DIX
= lokale Composition-/Application-Graphen
+ explizite Functions
+ optionale First-party-Module
+ zielsystemspezifische Delivery-Artefakte

ROBA
= eigenständiger ephemerer Context-State-Dienst
+ Contexts und Instances
+ Token- und SocketPrincipals
+ Resource ACLs
+ Daemon-, Control- und Context-Endpunkte
```

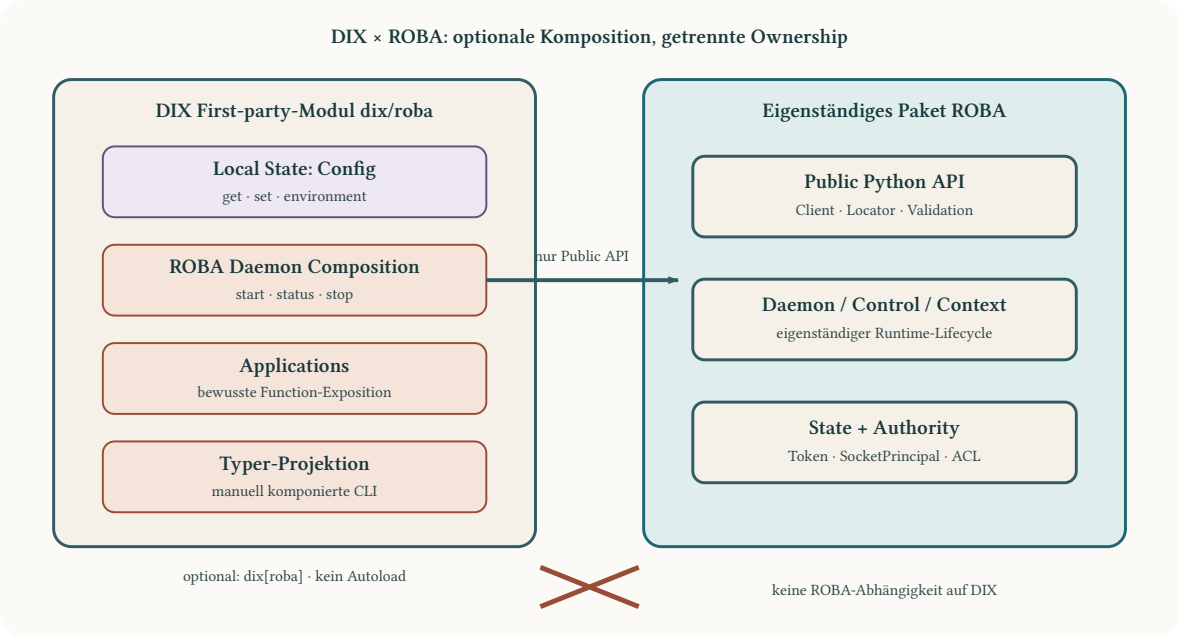

Abbildung 6: Die implementierte Abhängigkeitsrichtung: Das optionale DIX-Modul komponiert lokale Config, ROBA-Public-API, Applications und CLI. ROBA kennt DIX nicht.

Die Abhängigkeitsinvariante aus V1 blieb bestehen:

```
DIX ROBA Module → öffentliche ROBA API
ROBA Core       -X→ DIX
```

Das Packaging macht die Grenze sichtbar. DIX selbst besitzt keine ROBA-Basisdependency. Erst `dix[roba]` zieht das unabhängige Paket und die für dieses Modul benötigten optionalen Dependencies ein. Das Modul autoloadet nicht. Ein Consumer lädt es wie jedes andere DIX-Modul explizit. [E-043]
Diese Richtung ist wichtiger als die konkrete CLI. Würde ROBA nun DIX-Typen, DIX-Graphen oder DIX-Launcher kennen müssen, wäre keine Integration entstanden, sondern gegenseitige Assimilation.

8.4 UP-018 — Die erste reale Integrationsnaht

8.4.1 Invocation-lokale Konfiguration statt geerbter Umgebung

Die Composition `config` hält ein vollständiges lokales Modell für genau eine ROBA-Daemonkonfiguration. Sie exponiert drei schmale Functions:

```
get()                → vollständige Konfiguration
set(full_mapping)    → validieren und vollständig ersetzen
environment()        → explizite Prozessumgebung
```

Sie liest weder eine implizite `roba.toml` noch geerbte `ROBA_*`-Variablen. Beim Bau der Prozessumgebung werden vorhandene `ROBA_*`-Werte sogar entfernt und nur die lokal modellierten Werte neu gesetzt.
Damit ist die Konfiguration nicht global „DIX-State“. Sie gehört einer konkreten Composition-Instanz. Zwei DIX-Instanzen können zwei ROBA-Runtimes adressieren, ohne ihren Konfigurationszustand versehentlich zu teilen.

8.4.2 Adapter über die öffentliche Python-API

Die Composition `daemon` verwendet ROBA nicht über dessen CLI und nicht über interne Socketdetails. Sie adaptiert die öffentliche Python-API für genau drei Operationen:

```
start()
status()
stop()
```

Die darüberliegende DIX-Application exponiert diese Functions bewusst. Die Typer-Application wiederum projiziert sie in eine scriptbare CLI. Jede Schicht bleibt ein normaler Consumer der vorherigen:

```
ROBA Public API
    ↑
daemon Composition
    ↑
daemon Application
    ↑
Typer Application
```

Es gibt keinen neuen universellen Runner im Core. Auch die CLI wurde nicht aus einem globalen Contract-System magisch erzeugt. Sie ist manuell aus konkreten Applications zusammengesetzt — genau wie es der reduzierte B-013-Core vorsieht.

8.4.3 Was der Lauf belegt

UP-018 prüfte:

- optionales Packaging über Source und Wheel;
- Instanzisolation der Konfiguration;
- Entfernung geerbter ROBA-Variablen;
- Start, Status und Stop über ROBA Public API;
- Application- und Typer-Projektion;
- Fehlerpfade und Cleanup;
- unveränderte ROBA-Core-Tests.

Der technische Zielstand `b03b447` lief mit 141 DIX-Tests sowie 59 ROBA-Tests und 32 ROBA-Subtests grün. In den archivierten Outputs wurden Tokens nur als Fingerprints, nicht als Secrets, ausgegeben. Das belegt den Slice; es belegt weder Produktionssicherheit noch seine menschliche Abnahme. [E-043]

8.5 UP-019 — Von Daemonbedienung zu ephemerer Capability-Registry

Nach dem ersten Slice blieb ein konkreter Langläuferdruck offen: Ein Prozess A kann einen ROBA-Daemon und Context erzeugen, terminieren und später durch Prozess B ersetzt werden. Wenn alle Owner- und Control-Credentials nur im ersten Prozess leben, ist der technisch laufende Daemon praktisch nicht mehr verwaltbar.
UP-019 löst das nicht mit einer dauerhaften Secret-Datenbank. Es baut innerhalb des laufenden ROBA-Daemons eine ephemere Registry aus den bereits vorhandenen ROBA-Primitiven.

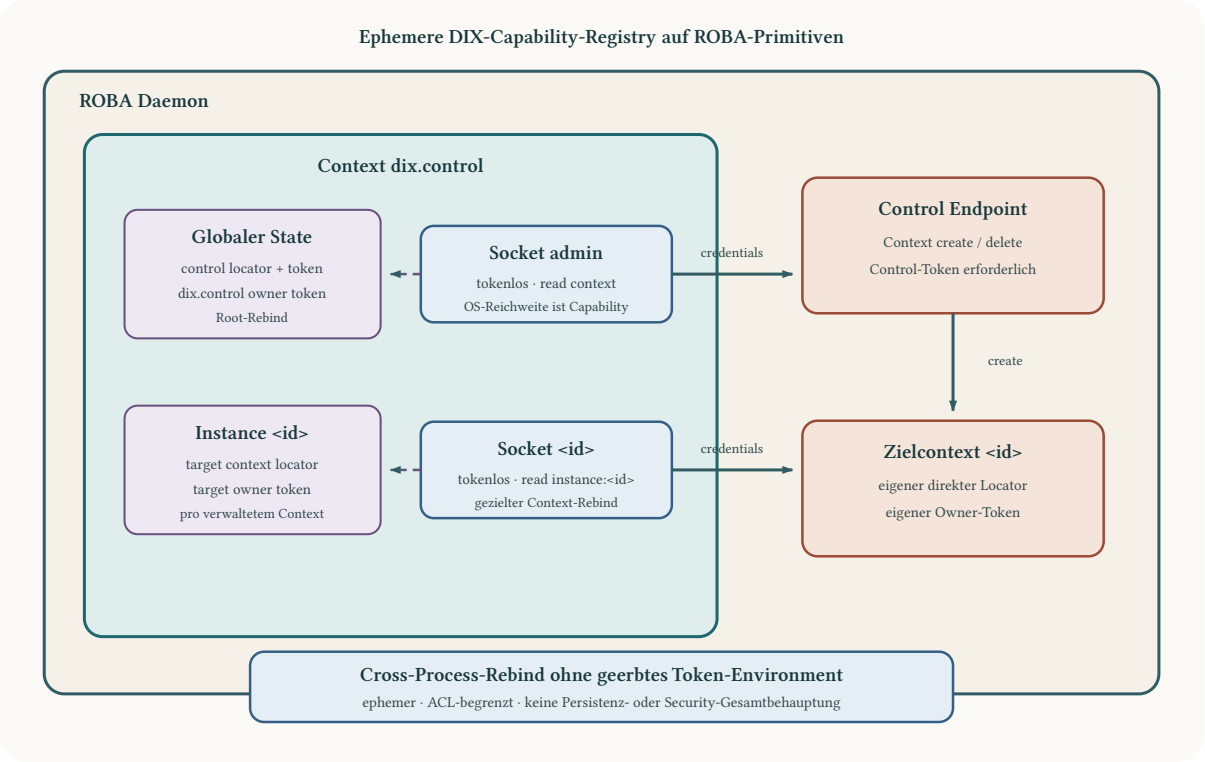

Abbildung 7: Die DIX-eigene Capability-Registry: Ein Control-Context hält Root- und Context-Credentials; tokenlose, per ACL begrenzte Sockets erlauben gezielten Rebind.

8.5.1 Topologie

Beim Bootstrap entsteht:

```
ROBA Daemon
└── Context dix.control
    ├── globaler State
    |   ├── control locator
    |   ├── control token
    |   └── owner token von dix.control
    ├── Socket admin
    |   └── read auf Context-State
    └── pro verwaltetem Context <id>
        ├── Instance <id>
        |   ├── direkter Context-Locator
        |   └── Owner-Token des Zielcontexts
        └── Socket <id>
            └── read ausschließlich auf Instance <id>
```

Der Root-Socket und die Manager-Sockets sind SocketPrincipals. Wer den jeweiligen Unix Socket erreicht, braucht für diese begrenzte Operation kein zusätzliches Token. Die Capability liegt in der Erreichbarkeit des Sockets plus seiner serverseitigen ACL.

Das ist bewusst keine Aussage, dass jeder lokale Socket sicher sei. OS-Rechte, Mounts, Namespacegrenzen und Socketverteilung bleiben äußere Authority-Entscheidungen. ROBA erzwingt nur die Resource-ACL, die der Socket repräsentiert.

8.5.2 Vier sichtbare Functions

Die Control-Composition exponiert vier zentrale Operationen:

```
bootstrap()
control_credentials()
create_context(context_id, name="")
context_credentials(context_id)
```

`bootstrap()` startet einen frischen Daemon, erzeugt `dix.control`, schreibt die Root-Credentials in dessen globalen State und erzeugt den read-only Admin-Socket.
`control_credentials()` liest diese Credentials über den tokenlosen Admin-Socket wieder aus. Ein späterer Prozess muss das einmalige Bootstrap- Ergebnis also nicht im Environment geerbt haben.
`create_context()` verwendet die wiedergewonnene Control-Authority, erzeugt einen Zielcontext und legt dessen Locator und Owner-Token in einer gleichnamigen Registry-Instance ab. Danach erzeugt es einen Manager-Socket, der nur diese Instance lesen darf.
`context_credentials()` bindet sich ausschließlich über diesen contextbezogenen Socket zurück. Der Consumer erhält damit Zugriff auf genau das registrierte Ziel, ohne zugleich die Root-Registry lesen zu können.

8.5.3 Warum das kein State-Mirroring im alten Sinn ist

Die Registry hält Credentials und Locator als expliziten Betriebsstate. Sie spiegelt nicht fortlaufend den fachlichen State der Zielcontexts. Es gibt keinen Sync-Loop und keinen Anspruch, die zweite Wahrheit über deren Inhalte zu sein.
Trotzdem ist eine wichtige Einschränkung nötig: Die Registry ist nur so lange autoritativ, wie der Daemon lebt und die Einträge nicht außerhalb ihres Managementpfads verändert werden. Sie übernimmt keine Haltbarkeitsgarantie. „Autoritativ“ heißt hier **für diesen laufenden Capability-Rebind**, nicht „dauerhaft über Prozess- oder Maschinenverlust“.

8.5.4 Was der Lauf belegt

Der technische Stand `f11a60b` bestand zehn fokussierte Registry-Tests, 147 DIX-Tests und einen echten Cross-Process-E2E. Prozess A erzeugte Daemon, Registry und Context; ein separater Prozess B gewann Credentials über die Capability-Sockets zurück. ROBA blieb auf seinem referenzierten Commit mit 59 Tests und 32 Subtests grün. [E-044]

8.6 RV-001 — Warum 147 grüne Tests nicht das Ende waren

Der unabhängige Review prüfte nicht nur Funktionsoutput, sondern Ticketgrenzen, Rollback und Operations-Evidence. Er fand drei verschiedene Problemarten.

8.6.1 1. Eine Ticketgrenze war verletzt

DX-072 sollte die Root-Registry bauen. Der Commit enthielt jedoch bereits wesentliche Managed-Context-Funktionalität aus DX-073. Das Endsystem konnte funktionieren; die behauptete Commit-Evidence war trotzdem falsch geschnitten.
Das ist kein Stilproblem. Wenn ein Commit mehr Architekturverantwortung trägt als sein Ticket, kann später weder Regression noch Review eindeutig sagen, welche Änderung welchen Claim beweist.

8.6.2 2. Ein Rollbackpfad war nicht vollständig

Beim Erzeugen eines Capability-Sockets wurde der Cleanup-Marker erst nach der erfolgreichen Rückgabe von `socket_create()` gesetzt. Damit blieb ein entscheidender Zwischenzustand unmodelliert:

```
Server hat Socket erzeugt
→ Antwort erreicht Client nicht
→ Call wirft Fehler
→ lokaler Code glaubt: Socket wurde nie erzeugt
→ Rollback lässt Ressource zurück
```

Der Fix setzt den Versuch **vor** dem Call als begonnen. Im Fehlerfall wird die Löschung versucht; ausschließlich ein nachweisliches `not found` darf ignoriert werden. Andere Cleanupfehler bleiben sichtbar. [E-045]

8.6.3 3. Der Evidence-Prozess produzierte selbst Fehler

Der erste Review-Run war ungültig, weil ein unquoted Heredoc Backticks als Shellkommandos interpretierte und damit Metadaten beschädigte. Der Run wurde nicht überschrieben, sondern mit `FAILED.md` erhalten. Run 002 belegte danach fünf Reviewtests, 148 DIX-Tests sowie die unveränderten ROBA-Suites.

Beim ersten Commit dieser Evidenz gelangte außerdem ein verstecktes Staging-Verzeichnis in den Commit. Die Ursache war banal und lehrreich: `git diff --cached --check` war nicht fail-fast mit dem Commit verkettet. Ein Folgecommit entfernte die Fläche wieder, statt die Historie zu glätten. Ein dritter Lauf führte anschließend genau den korrigierten Builder erneut gegen dieselben fixierten DIX- und ROBA-Commits aus. Alle zehn Exit-Artefakte standen auf `0`; ein Staging-Verzeichnis blieb nicht zurück. [E-046]

8.7 Was bestätigt, korrigiert und offen ist

8.7.1 Bestätigt

- DIX kann ROBA optional und ausschließlich über dessen Public API konsumieren.
- Die Konfiguration kann composition-lokal und frei von geerbter ROBA-Policy bleiben.
- Daemon-Lifecycle lässt sich durch normale DIX-Compositions, Applications und Typer-Projektion führen.
- Eine ephemere Capability-Registry kann mit ROBA-State, Instances und SocketPrincipals ohne zusätzlichen ROBA-Core-Mechanismus gebaut werden.
- Cross-Process-Rebind funktioniert technisch.

8.7.2 Korrigiert

- Der erste relevante Slice war Runtimekonfiguration und Daemonbedienung, nicht das willkürliche Lesen und Schreiben eines fachlichen Werts.
- „Run vorhanden“ genügt nicht; der Run selbst kann ungültig sein.
- „Alle Tests grün“ genügt nicht; Ticketgrenzen und ambige Cleanup-Zustände brauchen eigenen Druck.

8.7.3 Offen

- B-017, UP-018, UP-019 und RV-001 warten am Snapshot auf menschliche Abnahme. [E-048]
- Der geplante native bwrap-E2E mit gezielt gemountetem SocketPrincipal ist nicht Teil dieses Stands.
- Security-Audit, Remote-Transport, Windows-native Authority und dauerhafte Recovery bleiben außerhalb des Claims.
- Die DIX-README enthält an einer Stelle noch ältere Zukunftssprache zur ROBA-Integration; Code, Modul-README und Tests zeigen den neueren Stand.

8.8 Der noch offene Sandbox-Drucktest

Der stärkste nächste native Slice bleibt ähnlich wie in V1, nun aber auf realer Registry-Evidenz aufgebaut:

1. Parent besitzt Root- oder Context-Authority.
2. Er erzeugt einen SocketPrincipal mit minimaler ACL.
3. bwrap erhält nur Projekt-CWD und diesen Socket.
4. Der Child besitzt kein Owner-/Control-Token.
5. Eine DIX-Application im Child führt erlaubte Operationen aus.
6. Zugriff außerhalb der ACL scheitert unterscheidbar.
7. Socketentzug und Cleanup werden unter echten Fehlerpfaden beobachtet.

Linux-bwrap, Docker/OCI und Windows dürfen dafür weiterhin eigene native Module besitzen. Eine spätere kleine Compatibility-Function kann gemeinsame Fälle projizieren. Sie ist aber Ergebnis realer Übereinstimmung, nicht Voraussetzung der Integration.

8.9 Was dieser Vertikalschnitt methodisch beweist

V1 konnte nur argumentieren, dass zwei Public APIs begrifflich zusammenpassen. V1.1 besitzt jetzt stärkere, aber weiterhin begrenzte Evidenz:

```
Hypothese
→ optionales Packaging
→ lokaler Config-Owner
→ Public-API-Adapter
→ Application + CLI
→ ephemere Capability-Registry
→ Cross-Process-Rebind
→ unabhängiger Rollback-/Boundary-Review
```

Gerade die Abweichung vom alten ersten Slice ist der Gewinn. ACDD verlangt nicht, dass ein früher Zielwurf recht behält. Es verlangt, dass wir sichtbar machen, welcher reale Druck ihn verändert hat, was nun technisch existiert und welche Bedeutung noch auf Abnahme wartet.

9 AI-Kooperation, Firma und öffentliche Vermittlung

9.1 AI ist Beschleuniger und Fehlerverstärker

AI kann in einem Architekturzyklus große Mengen Kontext verarbeiten, Alternativen formulieren, Code implementieren und Evidenz zusammentragen. Sie kann ebenso schnell eine falsche Abstraktion konsistent durch das gesamte System ziehen. Deshalb braucht AI keine mystische Sonderrolle, sondern klare Authority- und Evidence-Grenzen.

9.1.1 Der Mensch verantwortet

- gewünschte Bedeutung und Priorität;
- legitime organisatorische Policy;
- Risikoakzeptanz;
- Entscheidung bei echten semantischen Forks;
- finale manuelle Abnahme;
- Freigabe interner oder öffentlicher Claims.

9.1.2 AI kann verantwortlich ausführen

- Bestand und Git-Historie prüfen;
- widersprüchliche Contracts finden;
- Gegenpositionen stark formulieren;
- Zielwürfe auf Vollständigkeit prüfen;
- Umsetzungspakete und Tickets aus bestätigter Semantik ableiten;
- kleine Tickets implementieren und committen;
- Tests und isolierte Artifact Runs ausführen;
- Evidence-/Claims-Ledger pflegen;
- Dokumentation gegen Code und Testoutput auditieren.

9.1.3 AI darf nicht still entscheiden

- ob ein neues Verhalten Core oder Module gehört, wenn Ownership offen ist;
- ob ein Replacement Backwards-Kompatibilität brechen darf;
- ob eine Sicherheitsgrenze akzeptabel ist;
- ob persönliche Policy allgemeiner Standard wird;
- ob fehlende Abnahme als bestanden gilt;
- ob eine neue öffentliche Behauptung aus indirekter Evidenz folgt.

9.2 Der Goal-Prompt ist Execution Contract, nicht Wahrheit

Ein guter Goal-Prompt ist detailliert, weil er Boundary, Scope, Ticketfolge, Tests und Completion Audit fixiert. Seine Länge verleiht ihm keine fachliche Autorität. Die Autoritätskette bleibt:

```
bestätigter Architekturstand
→ Umsetzungspaket
→ Goal-Prompt
→ Code und Evidenz
```

Entdeckt ein Agent einen Widerspruch zwischen Prompt und Block, muss er stoppen oder zum Block zurückkehren. „Autonom bis M1 durchrennen“ gilt für Implementationsarbeit innerhalb geklärter Semantik, nicht für das heimliche Erfinden der Semantik.

9.3 Prompt Engineering ersetzt keinen Standard

Eine wiederholte Instruktion wie

> „Nutze immer named parameters und erkläre alle Optionen inline“

ist nützlich, aber schwach:

- sie kann vergessen oder gekürzt werden;
- Modelle interpretieren sie unterschiedlich;
- sie ist nicht automatisch regressionstestbar;
- Abweichung wird erst im Review sichtbar.

Ein ausführbarer Standard kann dieselbe Regel in CLI-Deskriptor, Generator, Validator und Test ausdrücken. B-006 verwendet genau diesen Gedanken für ein opinionated Typer-Modul. Der Prompt fordert dann nur noch, das Standardartefakt zu verwenden und seinen Output zu prüfen. [E-015, E-016]

> **GEGENPOSITION** — Noch unstabile Regeln gehören nicht sofort in Tooling. Ein Prompt oder Block ist für Exploration billiger. Erst wiederholte, verstandene Semantik sollte ausführbarer Standard werden.

9.4 Inspectability statt Vertrauen in den Agenten

Ein AI-gestützter Lauf sollte von außen rekonstruierbar sein:

- welche Quellen und Commitstände wurden verwendet?
- welcher Zielwurf war bindend?
- welche Dateien änderten sich?
- welche Commits tragen welches Inkrement?
- welche Tests liefen mit welchem stdout?
- welches isolierte Artefakt entstand?
- was wurde manuell geprüft?
- welche offenen Punkte blieben?

Transkripte können ergänzen, sind aber schlechte Primärdokumentation. Sie sind lang, enthalten Zwischenannahmen und vermischen Frage, Interpretation und Entscheidung. Der Block destilliert die fachliche Entwicklung; Git und Tests belegen die technische.

9.5 Der Prozess muss ohne AI verständlich bleiben

Wenn nur das ausführende Modell erklären kann, wie ein Zyklus funktioniert, ist der Prozess nicht inspectable. Ein Mensch muss anhand der Artefakte:

1. den Ausgangsdruck verstehen;
2. Alternativen nachvollziehen;
3. den gewählten Contract identifizieren;
4. Code- und Testevidenz finden;
5. Abnahmestatus prüfen;
6. den nächsten offenen Druck erkennen.

AI erhöht Durchsatz. Sie ersetzt keine fachliche Lesbarkeit.

9.6 Fallstudie: der kleine Auftrag und die perfekte Plattform

Der Ausgangsdruck von DIX war kein Plattformmandat. Eine kleine interne Anwendung brauchte einen belastbaren Weg von formularnahen Eingaben zu Aktionen. Ein älterer Proof of Concept konnte als Erfahrungsmaterial dienen; der neue Bestand sollte sauberer, eigenständig und später möglicherweise wiederverwendbar werden. Diese Beschreibung ist absichtlich anonymisiert. Für die Architekturfrage zählen weder Auftraggeber noch fachliche Formulardaten. [S-15]

> **WERKSTATTSZENE · ANONYMISIERTE REKONSTRUKTION** S-15 · PARAPHRASE
>
> Die konkrete Lieferung war klein genug, dass man sie in einem lokalen Script hätte lösen können. Gleichzeitig waren die bekannten Reibungen groß genug, um sofort an Renderer, Actions, Config, Zustände, Wiederverwendung und mehrere Interfaces zu denken. Genau an dieser Stelle kann Architekturarbeit entweder verschwinden oder den Auftrag verschlucken.

9.6.1 Gefahr eins: Der Auftrag wird Vorwand für eine Plattform

Ein technisch ambitioniertes Team erkennt wiederkehrende Probleme oft früher als die Organisation bereit ist, sie als Produkt zu tragen. Das kann Stärke sein. Es wird zur Falle, wenn der konkrete Auftrag still für Forschung, Generalität und künftige Nutzer bezahlen soll.
Typische Warnzeichen:

- Der erste reale Consumer wartet auf ein generisches Pluginmodell.
- Nicht vorhandene zweite Backends bestimmen bereits den Core.
- „Später wiederverwendbar“ ersetzt einen benannten Product Owner.
- Akzeptanzkriterien prüfen Plattformmechanik statt den gelieferten Nutzen.
- jede lokale Besonderheit wird als Option in den allgemeinen Contract aufgenommen;
- Scopewachstum wird als unvermeidliche Architekturqualität dargestellt.

Eine elegante Plattform, die den kleinen Auftrag zu spät oder zu riskant liefert, ist kein Architekturgewinn. Sie ist ein anderes Vorhaben ohne ehrliche Budgetentscheidung.

9.6.2 Gefahr zwei: Wiederkehrende Reibung wird jedes Mal lokal bezahlt

Das Gegenextrem wirkt geschäftlich nüchtern: nur den Auftrag lösen, nichts abstrahieren. Wenn aber jedes neue Tool erneut Optionsparser, Fehlerformat, Configauflösung, Berechtigung, Rendererbindung oder Instanzisolation erfindet, bezahlt die Organisation dieselbe Erkenntnis mehrfach. Die Kosten erscheinen nur nicht als Plattformprojekt. Sie verteilen sich auf Integrationsfehler, inkompatible Bedienung, Wartung und spätere Migration.
Das kleine `argparse`-Beispiel aus B-006 ist dafür ideal. Für ein einzelnes Tool war die Wahl schnell und korrekt. Erst neben vielen möglichen Tools wurde sichtbar, dass benannte Parameter, Environment, Helptext, JSON, Fehlerkanal und Exitcodes immer neu entschieden würden. Daraus folgt nicht „jede CLI muss DIX nutzen“. Es folgt: Wiederholt sich genau dieser Contractdruck, besitzt er möglicherweise eigenen Produktwert. [S-13]

9.6.3 Drei Spuren statt einer vermischten Roadmap

Die praktisch tragfähige Trennung lautet:

```
DELIVERY
  Was muss für diesen Auftrag zuverlässig funktionieren?

EVIDENCE
  Welche Reibung oder Invariante könnte über diesen Auftrag hinaus tragen?

PRODUCTIZATION
  Welche bewusst finanzierte, gepflegte und veröffentlichte Capability
  extrahieren wir tatsächlich?
```

Die Spuren sind verbunden, aber nicht gleichberechtigt im Tagesgeschäft. Delivery hat einen Auftrag, Termin, Risiko und eine konkrete Abnahme. Evidence hat zunächst nur die Pflicht, Beobachtungen verlustfrei und klein zu bewahren. Productization beginnt erst durch eine eigene Entscheidung.

9.6.3.1 1. Delivery

Für den formularnahen Ausgangsfall hätte der Delivery-Contract etwa lauten können:

- definierte Eingaben annehmen und validieren;
- genau benannte Aktionen ausführen;
- Fehler für den realen Consumer sichtbar machen;
- den gewählten Browser-/Serverpfad E2E prüfen;
- keine allgemeine Workflow Engine versprechen.

B-001 tat etwas Ähnliches: Config, Spec, Runtime-State, HTTP und HTML wurden als kleiner vollständiger Slice verbunden. Der UI-nahe Core war noch nicht die spätere Ontologie. Er durfte für diesen Zyklus trotzdem existieren, weil er einen realen Pfad lieferte und abgenommen werden konnte. [E-001]
Delivery schützt vor dem Satz: „Wir können erst liefern, wenn die perfekte Abstraktion fertig ist.“

9.6.3.2 2. Evidence

Während der Lieferung werden wiederkehrende Druckpunkte nicht sofort zur Bibliothek, sondern zu prüfbaren Notizen:

```
Beobachtung:
  Der HTML-Renderer besitzt Template- und Interaktionssemantik.

Wiederholungsindikator:
  Ein zweites Interface würde dieselbe Semantik anders darstellen.

Risiko:
```

```
  Verschieben wir Widgets in Core, bindet Darstellung die Fachcapability.

Nächster billiger Test:
  Rendererownership in einem begrenzten Slice trennen.

Status:
  EVIDENCE, noch kein Produktversprechen.
```

Evidence darf im Auftragsrepo, in einem Architekturblock oder in einem firmenweiten Register liegen. Sie braucht noch keine allgemeine API. Ihr Wert ist, dass das nächste Team nicht wieder bei einer vagen Erinnerung beginnt. Evidence schützt vor dem Satz: „Das hatten wir schon dreimal, aber niemand weiß mehr genau, warum es weh tat."

9.6.3.3 3. Productization

Eine wiederverwendbare Capability erhält eine eigene Entscheidung, wenn mindestens eine tragende Begründung vorliegt:

- mehrere reale Consumer erzeugen denselben Bedeutungsdruck;
- eine strategische Plattformentscheidung finanziert bewusst Vorleistung;
- Security, Compliance oder Betrieb verlangen einen einheitlichen Contract;
- der Integrationsaufwand ist messbar größer als die Pflege eines gemeinsamen Artefakts;
- ein vertikaler Pressure Test zeigt, dass die gemeinsame Semantik tatsächlich kleiner und stabiler als die nativen Systeme ist.

Dann braucht Productization eigene Ownership:

- Produkt-/Capability-Owner;
- Support- und Versionsgrenze;
- Trust- und Distributionmodell;
- Compatibilityentscheidung;
- unabhängige Tests und Artifact Runs;
- Migration oder bewusst keine Kompatibilität;
- Budget, das nicht heimlich im Ausgangsauftrag steckt.

Productization schützt vor dem Satz: „Weil der Code schon hier liegt, ist er jetzt automatisch unsere Plattform."

9.6.4 Ein Artefakt, drei Statuswerte

Der gleiche Code kann die Spuren nacheinander durchlaufen, ohne dass seine frühere Bedeutung umgeschrieben wird:

Zeitpunkt	Delivery	Evidence	Productization
erster Slice	konkrete Aufgabe funktioniert	erste Reibung dokumentiert	nicht beschlossen
zweiter Drucktest	bestehender Auftrag bleibt stabil	gemeinsamer Contract wird belastet	Exploration finanziert
extrahiertes Modul	Consumer migriert bewusst	Herkunft und Grenzen bleiben sichtbar	Owner, Version und Support gesetzt

Umgekehrt darf ein lokal entstandenes Artefakt dauerhaft lokal bleiben. Nicht jede gute Idee muss ein Produkt werden. Auch eine strategische Plattform darf vor dem zweiten Consumer beginnen — aber dann lautet die ehrliche Begründung „wir investieren bewusst", nicht „der kleine Auftrag erforderte es".

9.6.5 Konkreter Company-Flow

Ein leichter Firmenbaustein kann so aussehen:

```
auftrag/
├── delivery.md          Ziel, Scope, Akzeptanz, Termin
├── architecture.md      nur die für Delivery nötigen Contracts
└── evidence.md          mögliche wiederkehrende Druckpunkte

capabilities/
└── candidate-index.md   organisationsweite, noch unverbindliche Evidenz
```

```
productization/
└── <capability>/          eigener Block, Owner, Budget und Lifecycle
```

`evidence.md` darf drei Zeilen haben. Der Index darf eine Capability wieder verwerfen. Erst die Productization-entscheidung erzeugt Verpflichtung. Damit werden Architekturbeobachtungen organisationsweit lernbar, ohne jedes Kundenproblem zum Vorwand für die „perfekte Plattform“ zu machen.

> **GEGENPOSITION** — Zusätzliche Spuren können in kleinen Teams mehr Pflege als Nutzen erzeugen. Dann genügen Statusmarker in einer Datei. Entscheidend ist die semantische Trennung, nicht die Ordnerzahl.

9.6.6 Was DIX in dieser Fallstudie beweist — und was nicht

Für den ersten Buchsnapshot ist belegt, dass sechs kurze Zyklen aus dem konkreten Slice mehrere Verantwortungen extrahierten und zwei Semantiken verwarfen oder ersetzten. V1.1 ergänzt einen noch härteren Befund: Die danach abgenommene Norn-/Strand-Architektur wurde real weitergebaut und unter einem Generator-Gegenbeispiel erneut ersetzt. Auf dem kleineren Core entstanden anschließend Local State und eine optionale DIX×ROBA-Naht. Nicht belegt ist dadurch, dass DIX bereits die wirtschaftlich richtige Plattform für eine Firma ist, dass seine Pflegekosten niedriger sind oder dass andere Teams denselben Contract brauchen.
Die richtige öffentliche Aussage ist deshalb nicht „aus einem kleinen Auftrag entstand unsere universelle Plattform“. Sie lautet: Der Auftrag erzeugte einen realen Pressure Test; aus wiederkehrender Evidenz entstand ein eigenständiger Architekturversuch, dessen heutiger technischer und offener Stand überprüfbar ist.

9.7 Public Casebook und Company Handbook sind verschiedene Produkte

Ein einziges Dokument mit `if public`-/`if internal`-Schaltern wird schnell unlesbar und riskant. Sinnvoller ist eine gemeinsame fachliche Basis mit zwei Overlays.

9.7.1 Gemeinsame Basis

- ACDD-Begriffe;
- Block-/Package-/Ticket-/Evidence-Vertrag;
- generische Templates;
- öffentliche DIX-Architektur;
- reproduzierbare, redigierte Cases;
- allgemeine Rollen und Driftwarnungen.

9.7.2 Public Casebook

Ziel: nachvollziehbare fachliche Vermittlung ohne interne Daten.

Enthält:

- generische DIX-Cases;
- öffentliche Commitspannen;
- technische Alternativen und Replacements;
- reproduzierbare Artifact Runs;
- redigierte Testoutputs;
- klare Snapshot- und Open-Grenzen;
- keine privaten Operations-Rohstände, Tokens oder Firmendaten.

Ein Public Case sollte nicht behaupten „ACDD garantiert bessere Architektur“. Er sollte zeigen: Hier war die Annahme, hier der Slice, hier die Evidenz, hier die Korrektur. Leser:innen können die Beweiskette selbst bewerten.

9.7.3 Company Handbook

Ziel: verbindlicher interner Engineering- und Freigabeprozess.

Zusätzlich zur Basis:

- Rollen und RACI;
- System-/Datenklassifikation;
- Security- und Compliance-Evidence;
- interne Repository- und Signaturregeln;
- Definition of Ready/Implemented/Accepted;
- Review- und Freigabegrenzen;

- AI-Nutzungs- und Datenrichtlinien;
- interne Fallstudien;
- Incident-/Exception-Prozess;
- Aufbewahrung und Löschung von Transkripten/Snapshots.

9.8 Ein mögliches Firmenmodell

9.8.1 Rollen

Rolle	Entscheidung	Evidence
Product/Domain Owner	Zielnutzen, fachliche Abnahme	Use Case, Acceptance Notes
Architect	Contract, Invarianten, Replacement	Block, Decision, Review
Tech Lead	Umsetzbarkeit, Ticket-Schnitt	Package, Plan
Developer/Agent	Code innerhalb Contract	Commits, Tests
Security/Compliance	Trust-/Daten-/Control-Grenzen	Threat Model, Controls
Independent Reviewer	Claims gegen Evidence	Reviewprotokoll

9.8.2 Freigabegrenzen

- **Exploration:** keine Produktzusage; WIP-Artefakte klar markiert.
- **Target Agreed:** implementierbar, noch nicht real bewiesen.
- **Technically Implemented:** Code/Tests/Run vorhanden, keine Nutzerabnahme.
- **Accepted:** reale Abnahme und Claims-Audit abgeschlossen.
- **Public:** Redaction, Lizenz, Security und Claim Boundary geprüft.

9.8.3 Security-/Compliance-Evidence

ACDD ersetzt kein Securityverfahren. Es kann dessen Beweiskette integrieren:

```
Threat / Requirement
→ Architekturcontrol
→ Implementation
→ automatisierter Security Test
→ Deployment Evidence
→ manuelle/organisatorische Freigabe
```

Wichtig ist die Statusklarheit. Ein Unit Test für ACL-Parsing ist kein Beleg, dass Produktionsdateirechte korrekt gesetzt sind. Ein erfolgreicher Penetrationstest ist keine fachliche Abnahme.

9.9 Definitionen für die Firma

9.9.1 Definition of Ready

- fachlicher Druck und Owner benannt;
- Daten-/Securityklasse bekannt;
- Architekturblock stabil genug;
- Nicht-Ziele und Migration geklärt;
- testbarer vertikaler Slice;
- AI darf die Quellen sehen und bearbeiten, die der Goal-Prompt voraussetzt.

9.9.2 Definition of Accepted

- Code und Tests auf fixiertem Commit;
- isolierter Artifact Run;
- manuelle Abnahme durch berechtigte Rolle;
- Security-/Compliance-Nachweise, soweit erforderlich;
- Claims Ledger ohne ungekennzeichnete Inference;
- Rollout-/Rollbackentscheidung;
- öffentliche Kommunikation separat freigegeben.

9.10 AI-Nutzungsregeln als Company Overlay

1. **Capability-Logik:** Sichtbarkeit ist Freigabe; High-Value-Scopes werden separat bewertet.
2. **Keine versteckte Semantik:** Bei Architecture Fork zurück in den Block.
3. **Keine erfundene Evidenz:** stdout, Commit und Abnahme nur real erfassen.
4. **Snapshotdisziplin:** parallele Quelländerungen nicht still mischen.
5. **Secret-Hygiene:** Outputs und PDFs vor Veröffentlichung scannen.
6. **Inspectable Changes:** kleine, benannte Commits; keine kosmetische Mischänderung.
7. **Human Acceptance:** ein Agent darf technische Completion melden, aber Nutzerabnahme nicht simulieren.
8. **Model Independence:** Prozessartefakte müssen von einem anderen Menschen oder Tool lesbar bleiben.

9.11 Fachliches Lehrbuch aus realen Operations

Operations ist reichhaltiger als ein veröffentlichbares Buch. Es enthält:

- ungeschliffene Fragen;
- verworfene Zwischenstände;
- private Kontexte;
- lokale Pfade und Testfixtures;
- Wiederholungen;
- WIP ohne Commit.

Die redaktionelle Aufgabe ist **nicht**, alles zu kürzen, bis nur eine saubere Erfolgsgeschichte bleibt. Sie besteht aus vier Transformationen:

1. **Destillieren:** relevante Architekturbewegung aus Rohverlauf lösen.
2. **Verifizieren:** gegen Commit, Code, Test und Artifact Run prüfen.
3. **Klassifizieren:** Fakt, Inference, Interpretation, Open, Rejected, Replaced.
4. **Redigieren:** private Details entfernen, ohne die negative Evidenz zu glätten.

Das Resultat ist kein Marketingcase, sondern ein fachliches Casebook.

9.12 Ein ehrliches Public-Narrativ

Schlecht:

> „In fünf Tagen entstand mit AI eine revolutionäre modulare Plattform."

Besser für den ersten Snapshot:

> „Sechs eng dokumentierte Zyklen verschoben DIX von einem UI-nahen Foundation- Slice zu einem Capability-/Composition-/Application-Kern. Zwei zentrale Semantiken wurden bewusst verworfen oder ersetzt. Der Snapshot bis DX-029 hat 209 grüne Tests; der vollständige B-006-E2E und externe Übertragbarkeit sind offen."

Das zweite Narrativ ist weniger spektakulär und fachlich viel stärker. Es lädt zur Prüfung ein, statt Vertrauen zu verlangen. V1.1 darf ergänzen:

> „Der Folgezyklus implementierte und akzeptierte zunächst einen composition-scoped Contract-Core, ersetzte ihn nach einem konkreten Generator-Gegenbeispiel wieder und baute auf dem reduzierten Kern eine optionale ROBA-Integration. 148 DIX-Tests und der Cross-Process-E2E sind reproduziert; die menschliche B-017-/Package-Abnahme bleibt offen."

Auch dieses Narrativ ist kein Firmenbeweis: externe Anwendung, Kosten/Nutzen, Prozessvergleich, Wartungsbeobachtung und Security-/Compliance-Overlays bleiben **OPEN**. Anhang C führt diese Claim-Grenze aus.

10 Handbook: ACDD direkt anwenden

10.1 Minimaler Einstieg

Für den ersten Zyklus braucht ein Team nicht sofort ein neues Tool. Ein Repositoryordner genügt:

```
operations/
├── blocks/
├── decisions/
├── packages/
├── tickets/
├── test-cases/
├── artifact-runs/
└── evidence/
```

Die Ordner sind kein Dogma. Entscheidend ist die Richtung:

- während offener Semantik lebt die Wahrheit im Block;
- nach dem Zielwurf wird ein Umsetzungspaket abgeleitet;
- Tickets und Commits bilden beweisbare Schritte;
- Artifact Run und Abnahme schließen den Zyklus;
- Claims referenzieren Evidenz statt nur den Endzustand.

10.2 Rollen und Verantwortung

Rolle	Verantwortet	Darf nicht delegieren
Semantik-Owner	Bedeutung, Ziel, Nicht-Ziele, Akzeptanz	finale fachliche Entscheidung
Architekturpartner	Gegenpositionen, Grenzen, Zielwurfschärfung	Konsens vorspielen
Developer	Implementation, lokale Designentscheidungen im Contract	versteckte Semantikänderung
AI-Agent	Bestand prüfen, Varianten strukturieren, Tickets umsetzen, Evidenz sammeln	unklare Forks still entscheiden
Reviewer	Contract-/Code-/Evidence-Konsistenz	Abnahme ohne reales Artefakt
Operator/Security	Runtime-, Trust- und Deploymentgrenzen	fachliche Semantik aus Infrastruktur ableiten

In kleinen Teams kann eine Person mehrere Rollen tragen. Die Verantwortungen sollten trotzdem im Artefakt erkennbar bleiben.

10.3 Template 1 — Architekturblock

```
---
id: B-XXX
title: <kurzer Problemname>
status: exploration | target-agreed | implemented | accepted | replaced
opened: YYYY-MM-DD
source: <voriger Block, Incident, Use Case>
---

# B-XXX — <Titel>
```

```
## Ausgangsdruck
Welches beobachtbare Verhalten, Risiko oder reale Ziel erzeugt den Block?

## Aktueller Befund
- VERIFIED: ...
- INFERENCE: ...
- OPEN: ...

## Begriffe
Was bedeuten die strittigen Wörter in diesem Block konkret?

## Plausibles bisheriges Modell
Warum war/ist es vernünftig? Keine Strohpuppe bauen.

## Gegenpositionen und Alternativen
### A — ...
Nutzen, Kosten, Bruch.
### B — ...
Nutzen, Kosten, Bruch.

## Architekturgrenzen
- Owner:
- Source of Truth:
- Trust Boundary:
- Dependency Direction:
- Fehler-/Rollbackbehauptung:

## Aktueller Zielwurf
Präziser Contract in wenigen Regeln.

## Invarianten
- ...

## Nicht-Ziele
- ...

## Widerlegungsbedingungen
Welcher reale Befund würde den Zielwurf ersetzen oder schärfen?

## Offene Entscheidungen
- O-001 ...

## Verlauf
### YYYY-MM-DD — <Korrektur>
Was änderte sich und warum?

## Abnahme
Wer prüfte welches Artefakt mit welchem Ergebnis?
```

10.3.1 Block-Qualitätscheck

- Beschreibt der Ausgangsdruck ein Problem statt eine Lösung?
- Ist das bisherige Modell fair und plausibel dargestellt?
- Sind Alternativen echte Contracts statt Schlagwörter?
- Kann jemand die Source of Truth und Authority benennen?
- Ist klar, was bei Fehlern garantiert und was nicht garantiert wird?
- Gibt es mindestens einen widerlegbaren Befund?
- Wurde ein Replacement als solches markiert?

10.4 Template 2 — Semantische Schärfung

Diese Fragen können in Review oder AI-Kooperation verwendet werden:

10.4.1 Verantwortung

- Wer besitzt die Bedeutung dieses Begriffs?

- Kann die vorgeschlagene Schicht die Garantie vollständig kontrollieren?
- Ist das Mechanismus, Policy, UX oder Domain Behavior?
- Wird eine persönliche Arbeitsweise als allgemeine Semantik verkauft?

10.4.2 State

- Ist der State autoritativ oder eine Projektion?
- Wer darf ihn ändern?
- Was passiert bei parallelen Instanzen?
- Warum braucht der Consumer Daten statt einer engeren Function?
- Welche Staleness- oder Konfliktsemantik wird behauptet?

10.4.3 Funktionen und Verträge

- Welche Function muss ein Consumer wirklich aufrufen können?
- Woher stammt ihre autoritative Signatur?
- Ist ein Wrapper lokale Ownership oder nur Boilerplate?
- Was heißt unsupported?
- Ist ein Hook strukturell oder fachlich?

10.4.4 Trust und Authority

- Was ist Credential, was Locator, was Identität?
- Welche Prüfung schützt welche Ressource?
- Ist deklarative Konfiguration fälschlich als Sicherheitsgrenze dargestellt?
- Was kann fremder Code außerhalb des behaupteten Rollbacks verändern?

10.4.5 Zukunft

- Gibt es einen realen zweiten Consumer?
- Welche „künftige Flexibilität“ wird gerade nur vermutet?
- Würde ein natives Zielmodul ehrlicher sein?
- Welches Compatibility-Artefakt könnte später separat entstehen?

10.5 Template 3 — Gegenposition / Alternative

```
### Alternative A — <Name>

**Stärkster Grund dafür**
...

**Contract**
...

**Gewonnene Eigenschaften**
- ...

**Kosten / verlorene Eigenschaften**
- ...

**Realer Pressure Test**
...

**Widerlegender Befund**
...

**Warum aktuell angenommen / verworfen / offen**
...
```

Die Pflicht „stärkster Grund dafür“ verhindert Strohmänner. Eine Alternative ohne prüfbaren Contract ist nur ein Schlagwort.

10.6 Template 4 — Umsetzungspaket

```
---
id: UP-XXX
source_block: B-XXX
```

```
status: draft | ready | in-progress | implemented | accepted
---

# UP-XXX — <Titel>

## Verbindlicher Architekturstand
Kurze Referenz auf die bestätigten Blockabschnitte.

## Ziel
Welcher vertikale Slice entsteht?

## Public Contract Changes
- add ...
- replace ...
- remove ...

## Nicht-Ziele
- ...

## Geplante Ticketschnittfolge
1. TX-001 — ...
2. TX-002 — ...

## Automatisierte Akzeptanz
- Positivpfad
- Fehlerpfad
- Isolation
- Teardown
- Public Contract

## Isolierter Artifact Run
Inputs, Build, Runtime, Outputs, Reviewreihenfolge.

## Manuelle Abnahme
Konkrete Handgriffe und erwartete Beobachtung.

## Stop Conditions
- semantischer Fork;
- erforderliche versteckte Core-Ausweitung;
- nicht reproduzierbare Evidenz;
- Source-/Trust-Grenze unklar.

## Goal-Prompt
Ausführbarer Auftrag; keine neue Semantik.
```

10.7 Template 5 — Ticket

```
---
id: TX-XXX
package: UP-XXX
status: open | active | done | accepted
---

# TX-XXX — <beweisbares Inkrement>

## Contract Delta
Welche beobachtbare Aussage gilt nach diesem Ticket anders?

## Scope
- ...

## Nicht-Ziele
- ...
```

```
## Implementation Notes
Erlaubte lokale Freiheitsgrade; keine neuen Architekturentscheidungen.

## Tests
- ...

## Evidence Output
Welche Ausgabe/Datei/Commit belegt das Ergebnis?

## Commit
Hash und Subject nach Umsetzung.

## Offene Abnahme
Nur wenn das Ticket selbst manuell geprüft werden muss.
```

10.7.1 Gute und schlechte Tickets

Schlecht	Besser
„Models ändern“	„Reservierte Lifecycle-Felder aus Composition-/Application-Spec entfernen“
„Tests ergänzen“	„Structural rollback bei Constructorfehler ohne fachlichen Hook belegen“
„CLI bauen“	„Typer-Binding aus normalisierten Optionsdeskriptoren erzeugen“
„Cleanup“	„Module Unload bei lebender fremder Instanz hart ablehnen“

10.8 Template 6 — Goal-Prompt

```
# Goal: Implementiere UP-XXX bis zur technischen Evidenz

## Authority
- Architekturquelle: operations/blocks/B-XXX.md, Abschnitt ...
- Umsetzungspaket: operations/packages/UP-XXX.md
- Bei Widerspruch gilt der bestätigte Block; nichts still uminterpretieren.

## Boundary
- schreibbare Repos/Pfade:
- nur lesbare Quellen:
- verbotene Bereiche:
- keine globalen Installationen / keine Secret-Ausgabe.

## Muss-Ergebnis
- vollständige Ticketfolge;
- Commits pro beweisbarem Inkrement;
- Regression;
- isolierter Artifact Run;
- Evidence-Outputs;
- Statusupdates in Tickets/Package.

## Nicht-Ziele
- ...

## Semantische Stop Conditions
- neue öffentliche Verantwortung;
- Alternative mit anderem Owner/Source-of-Truth;
- Compatibilityanforderung ohne belegten Consumer;
- Änderung einer bestätigten Invariante.

## Arbeitsweise
```

```
1. Boundary und Baseline prüfen.
2. sichtbaren Plan erstellen.
3. Tickets nacheinander umsetzen und committen.
4. Tests und Output unverändert erfassen.
5. Artifact Run neu und isoliert bauen.
6. Completion Audit gegen alle Akzeptanzkriterien.

## Abschlussbericht
Commits, Tests, Artifact-Pfad, offene Punkte, unveränderte Quellen.
```

Der Goal-Prompt sollte autonomes Arbeiten ermöglichen, aber semantische Stopps nicht wegoptimieren. „Entscheide alles selbst“ ist nur für redaktionelle und technische Freiheitsgrade legitim.

10.9 Template 7 — Evidence-/Claims-Ledger

```
| ID | Status | Claim/Befund | Source | Grenze |
|---|---|---|---|---|
| E-001 | VERIFIED | ... | file#section, commit, test | ... |
| E-002 | INFERENCE | ... | E-001 + E-003 | ... |
| E-003 | OPEN | ... | ticket / missing run | ... |
```

10.9.1 Statusregeln

- `VERIFIED` nur bei direkt prüfbarem Artefakt.
- `INFERENCE` nennt die zugrunde liegenden Belege.
- `INTERPRETATION` wird nicht als Fakt formuliert.
- `OPEN` bleibt offen, auch wenn die geplante Lösung überzeugend klingt.
- `REJECTED` beschreibt eine echte explorierte Alternative.
- `REPLACED` benennt alten und neuen Contract.

10.9.2 Claims-Audit

Für jede zentrale Aussage:

1. Welche genaue Evidence-ID trägt sie?
2. Belegt die Quelle Verhalten, Absicht oder Abnahme?
3. Ist der zeitliche Snapshot passend?
4. Wird aus einem Beispiel unzulässig Universalität abgeleitet?
5. Fehlt eine plausible Gegenposition?

10.10 Template 8 — Artifact Run

```
artifact-runs/B-XXX/run-001/
├── README.md
├── COMMANDS.md
├── FLOW.md
├── inputs/
├── project/
├── outputs/
|   ├── command-output/
|   ├── inspection/
|   └── runtime/
├── run.sh
├── tree.txt
└── SHA256SUMS
```

10.10.1 Anforderungen

- Runziel existiert vorher nicht; kein stilles Überschreiben.
- Code-Commit/Dependencyversion ist fixiert.
- Inputs sind vollständig oder referenzieren immutable Artefakte.
- jeder Command nennt Input und Output;
- Runtime-Smoke prüft mehr als nur Build;
- der Run verändert keine Quellrepositories;
- sensible Werte werden redigiert;

- Wiederholung erzeugt einen neuen Run oder beweisbar identischen Output.

10.10.2 Artifact-README

```
# B-XXX — Run 001

Code-Stand: <commit>
Built at: <timestamp>
Builder: ./run.sh

## Claim under pressure
...

## Review path
1. input/spec
2. generated artifact
3. inspection output
4. runtime result
5. teardown proof

## Intentionally absent
- ...

## Known nondeterminism
- timestamp ...
```

10.10.3 Wenn der Builder selbst scheitert

Ein angelegter Run-Ordner wird nicht nachträglich in einen erfolgreichen Run umgeschrieben. Stattdessen:

```
run-001/
├── FAILED.md          Ursache, Exitcode, betroffene Outputs
├── raw/               vorhandene Rohoutputs
└── ...

run-002/               vollständig neuer Beleg
```

`FAILED.md` muss klar sagen, ob der Produktclaim widerlegt wurde oder nur die Evidence-Erzeugung ungültig war. Ein geplatzter Heredoc, ein falscher Checkout oder fehlende Git-Provenienz sind keine Produktregression — aber sie verbieten, den Run als Beleg zu verwenden.

10.11 Template 8b — Unabhängiger Boundary-/Rollback-Review

```
# RV-XXX — Review <Slice>

## Authority und Source
- Zielblock / Package:
- exakter Commitbereich:
- vorhandener Artifact Run:

## Reviewfragen
- Trägt jeder Ticketcommit nur seinen behaupteten Contract?
- Welche Mutationen können serverseitig erfolgreich und clientseitig
  fehlgeschlagen erscheinen?
- Ist Cleanup vollständig, idempotent und fehlertransparent?
- Belegt der Run wirklich seinen Source-Stand?
- Sind Secrets, WIP und Stagingfläche ausgeschlossen?

## Findings
### F-01 — <Titel>
Severity:
Evidence:
Claim impact:
Required correction:
```

```
## Reproduzierbarer Reviewlauf
- neue fokussierte Tests;
- volle Regression;
- exakte Sourcematerialisierung;
- Rohoutput und Prüfsummen;
- negative Ergebnisse bleiben sichtbar.

## Ergebnis
PASS | FIXED | OPEN | INVALID_EVIDENCE
```

Der Review muss eine neue Angriffslinie besitzen. Eine bloße Wiederholung des Featurebuilders ist Regression, kein unabhängiger Review.

10.12 Template 9 — Manuelle Abnahme

```
# Acceptance — B-XXX / Run 001

Reviewer:
Zeitpunkt:
Artefakt-Commit/Hash:

## Preconditions
- frischer Run;
- keine lokalen versteckten Configs;
- erwartete Toolversionen.

## Schritte und Beobachtungen
1. ...
   - erwartet:
   - beobachtet:
   - Status: PASS | FAIL | QUESTION

## Semantische Fragen
- Besitzt die richtige Schicht das Verhalten?
- Ist etwas überraschend implizit?
- Bleibt Fehlerursache sichtbar?
- Funktioniert ein zweiter realer Input/Consumer?
- Ist ein Nicht-Ziel versehentlich eingezogen?

## Ergebnis
ACCEPTED | SHARPEN | REPLACE | OPEN

## Begründung
...
```

Die Abnahme darf nicht nur Expected Outputs abhaken. Mindestens eine Frage muss die Architekturbehauptung selbst belasten.

10.13 Template 10 — Replacement Decision

```
# Replacement R-XXX — <alter Contract> → <neuer Contract>

## Alter realer Contract
APIs, States, Hooks, Verhalten, Commitspanne.

## Warum er plausibel war
Stärkste fachliche und technische Gründe.

## Widersprechende Evidenz
Artifact Run, Observation, User Acceptance.

## Semantischer Bruch
Welche Verantwortung / Source of Truth / Authority war falsch?
```

```
## Neuer Contract
Präzise Replacement-Regeln.

## Entfernen
- APIs
- Flags
- States
- Fixtures
- Dokumentannahmen

## Bewahren
- Mechanik
- Tests als Regression anderer Invarianten
- Generator/Inspection
- Migrationswissen

## Compatibility Decision
Keine | Migration | Versionierter Legacypfad bis Datum X

## Neuer Pressure Test
...
```

10.14 Template 11 — Cycle-Abschluss

```
# Cycle Close — B-XXX

## Outcome
CONFIRMED | SHARPENED | EXTENDED | REPLACED | ABORTED | OPEN

## Evidence Chain
Block → Package → Tickets → Commits → Tests → Run → Review → Acceptance

## Stabil gewordene Invarianten
- ...

## Entfernte Annahmen
- ...

## Noch offene Unsicherheit
- ...

## Neuer Druck / Folgeblock
- ...

## Public Claim Boundary
Was darf extern behauptet werden, was nicht?
```

10.15 Template 12 — Delivery / Evidence / Productization

Für kleine konkrete Aufträge darf dieser Split in einer einzigen Datei leben:

```
# Delivery

## Konkretes Ergebnis
Was muss für diesen Auftrag funktionieren?

## Akzeptanz
Wer prüft welchen realen Pfad?

## Nicht-Ziele
Welche Plattform-/Wiederverwendungsfragen blockieren die Lieferung nicht?

# Evidence
```

```
## Wiederkehrender Druck
Welche Reibung könnte über den Auftrag hinaus relevant sein?

## Belege
Wo trat sie konkret auf? Keine Zukunftsvermutung als Wiederholung ausgeben.

## Nächster billiger Test
Welcher zweite Consumer oder Slice könnte die Gemeinsamkeit widerlegen?

## Status
OBSERVED | REPEATED | STRATEGIC | REJECTED

# Productization Decision

Status: NOT_DECIDED | EXPLORE | BUILD | REJECT
Owner:
Budget/Zeitraum:
Consumer:
Public Contract:
Support-/Versionsgrenze:
Migration:
```

`NOT_DECIDED` ist ein legitimes Ergebnis. Die Evidence geht dadurch nicht verloren; der Delivery-Auftrag erhält aber auch keine versteckte Plattformpflicht.

10.16 Reale Gespräche als Evidence verwenden

Transkripte können zeigen, *wann* ein Problem erkannt wurde und *warum* eine Richtung plausibel war. Sie sind keine technische Autorität. Für ein source-backed Casebook gilt:

Klasse	zulässige Darstellung
VERBATIM	exakter kurzer Wortlaut; Kürzung sichtbar; Event referenziert
PARAPHRASE	inhaltlich belegte Verdichtung ohne Anführungszeichen
RECONSTRUCTION	mehrere belegte Stellen; keine perfekte Echtzeitfolge behaupten
INTERPRETATION	heutige Deutung ausdrücklich markieren
COMPOSITE	Lehrbeispiel aus mehreren Fällen offen kennzeichnen
FICTIONAL	nur als Pseudobeispiel, nie als historische Szene
OPEN	attraktive, aber nicht zuverlässig belegbare Erinnerung weglassen

10.16.1 Minimaler Quote-Audit

```
Quote-ID
→ Session-/Event-ID
→ unveränderter Rohsnapshot + SHA-256
→ exaktes Fragment
→ Manuskriptstelle
→ Privacy-/Kontextprüfung
```

Sichtbare Assistant-Antworten können Quelle sein. Verdecktes Model-Reasoning darf nicht nacherzählt werden. Terminaltext in einer Usernachricht wird als sichtbarer Transkriptbeleg behandelt, technische Claims daraus aber gegen den eigentlichen Test-/Codesnapshot geprüft.

10.17 Kooperatives Semantikreview in fünf Zügen

1. **Owner formuliert Druck**, noch ohne gewünschte Lösung.
2. **Partner baut die stärkste Gegenposition**, einschließlich realer Vorteile und Kosten.
3. **Owner steelmantelt sie zurück:** Was daran stimmt, bevor er widerspricht.
4. **Beide verschieben die Ebene:** Geht der Streit um Mechanik, Ownership, Policy, Authority oder Delivery?
5. **Artefakt entscheidet nur seinen Claim:** Test und Run werden nicht zur simulierten Nutzerabnahme aufgeblasen.

Ein Review ist nicht besser, weil es konfliktfrei endet. Es ist besser, wenn der verbleibende Contract enger und seine Widerlegungsbedingung klarer ist.

10.18 Definition of Ready für ein Umsetzungspaket

Ein Package ist **ready**, wenn:

- Ausgangsdruck belegt ist;
- strittige Begriffe definiert sind;
- Owner, Source of Truth und Trust Boundary geklärt sind;
- Replacement versus Erweiterung explizit ist;
- mindestens eine ernsthafte Gegenposition bewertet wurde;
- Public Contract Delta benannt ist;
- Nicht-Ziele den Slice begrenzen;
- Akzeptanz- und Widerlegungsbedingungen existieren;
- der Ticket-Schnitt grün commitbar ist;
- keine offene Semantikfrage nur in den Goal-Prompt verschoben wurde.

10.19 Definition of Technically Implemented

- alle Tickets im Scope committed;
- Regression grün;
- Public Contract dokumentiert;
- isolierter Artifact Run gebaut;
- Evidence Outputs vorhanden;
- geforderter Boundary-/Rollback-Review ausgeführt oder ausdrücklich als nicht nötig begründet;
- keine uncommitted Quellabhängigkeit;
- keine behauptete manuelle Abnahme.

10.20 Definition of Accepted

- technically implemented erfüllt;
- Reviewfindings behoben, akzeptiert oder als offene Grenze klassifiziert;
- reales Reviewartefakt manuell geprüft;
- Abnahmebeobachtungen dokumentiert;
- Gegenprobe/zweiter Input, soweit relevant;
- Ergebnis `ACCEPTED` oder explizites `SHARPEN/REPLACE`;
- Claims Ledger aktualisiert;
- verbleibende Lücken und nächster Druck benannt.

10.21 Warnzeichen für Architekturdrift

- Implementation ergänzt einen neuen Begriff, der im Block nicht vorkommt.
- Goal-Prompt wird länger, weil der Contract selbst unklar bleibt.
- Tickets nennen Dateien statt beobachtbare Änderungen.
- Tests kennen nur Happy Paths der konkreten Demo.
- ein Adapter beginnt State zu besitzen.
- ein Reference Client wird Voraussetzung für andere Clients.
- neue Flags erhalten zwei widersprüchliche Grundsemantiken.
- „später“ enthält genau die Trust-/Parallelitätsgrenze, die den aktuellen Claim tragen müsste.
- WIP-Code wird in die Dokumentation als bereits committed eingemischt.
- Abnahme wird aus einem CI-Ergebnis abgeleitet.
- ein Ticketcommit implementiert bereits die Verantwortung des Folgetickets;
- Cleanup startet erst, wenn der mutierende Call erfolgreich zurückkehrte;
- ein fehlgeschlagener Artifact Run wird still repariert oder überschrieben;
- der Evidence-Builder prüft sich nur mit seinen eigenen Erfolgsannahmen.

10.22 Warnzeichen für vorschnelle Universalität

- abstrakter Name ohne zweiten realen Provider;
- gemeinsame Felder stammen vollständig aus dem ersten Tool;
- Unsupported wird als leerer Wert modelliert;

- Targetunterschiede landen als Optionssammlung;
- das Interface kann seine Fehlergarantie nicht für alle Backends halten;
- „flexibel“ bedeutet vor allem Templates, Includes und Overrides;
- die persönliche UX wird nur konfigurierbar, aber nicht entkoppelt;
- Compatibility sitzt im Core statt in einem separaten Artefakt.

10.23 Exploration oder implementierbarer Zielwurf?

10.23.1 Noch Exploration

- mehrere Alternativen unterscheiden sich in Ownership;
- Source of Truth ist offen;
- der gleiche Begriff meint verschiedene Dinge;
- Widerlegungsbedingung fehlt;
- der erste echte E2E ist nicht beschreibbar;
- Nicht-Ziele würden die zentrale Risikogrenze ausblenden;
- die vorgeschlagene API braucht versteckte implizite Logik.

10.23.2 Implementierbar

- ein Contract ist gewählt und begründet;
- Alternativen sind dokumentiert, nicht nur vergessen;
- Inputs, Outputs und Fehler sind beschreibbar;
- Trust/Authority ist klar;
- der Slice überquert relevante Schichten;
- Tickets können einzeln beweisbar committed werden;
- ein Artifact Run kann die Annahme real belasten.

10.24 Public-/Company-Redaction-Check

- keine Tokens, Secrets, Credentials;
- keine ungefilterten lokalen absoluten Pfade;
- keine persönlichen Namen ohne Zweck/Einwilligung;
- keine Kund:innen-, Mitarbeiter:innen- oder Firmendaten;
- keine Rohchats als bequemes Quellen-Dump;
- Commit- und Testclaims stimmen mit öffentlichem Snapshot;
- interne Controls nicht als öffentlich implementiert darstellen;
- offene Abnahmen bleiben offen;
- private Mappings/Pseudonymtabellen bleiben getrennt;
- Lizenz- und Third-party-Hinweise geprüft.

10.25 30-Minuten-Cycle-Review

Für kleine Teams reicht oft eine kurze feste Sequenz:

1. **5 min — Druck:** Welcher konkrete Befund startet den Zyklus?
2. **7 min — Semantik:** Owner, Wahrheit, Authority, Failure.
3. **5 min — Gegenposition:** stärkste Alternative und Widerlegung.
4. **5 min — Slice:** kleinster echter vertikaler Test.
5. **3 min — Nicht-Ziele:** was wird bewusst nicht gebaut?
6. **3 min — Evidence:** welche Outputs/Abnahme brauchen wir?
7. **2 min — Entscheidung:** implementieren oder weiter explorieren?

Die Zeitbox ist kein Ersatz für Tiefe. Sie verhindert nur, dass einfache Entscheidungen durch Artefaktpflege aufgeblasen werden.

11 Schluss: Architekturwissen entsteht im Widerspruch

Die Mounts gingen. Der Lifecycle ging. Die Tests gingen. Genau deshalb sind diese Fälle stark.
ROBA zeigt einen Mechanismus, der früh real funktionierte, während seine übergreifende Bedeutung noch jahrelang hätte weiterwachsen können. DIX zeigt einen vollständigen Contract, der gerade durch seinen sauberen Artefaktlauf semantisch zerbrach. In beiden Fällen wäre die einfache Geschichte falsch: Weder war alles vorher Unsinn, noch war der spätere Schnitt von Anfang an offensichtlich.
Die verwertbare Geschichte lautet:

- bwrap, Capability-Reduktion und dynamische Mount-Propagation blieben echte technische Evidenz;
- Hostpfadidentität verlor gegen inspectable Runtime Truth;
- persönliche Profile verloren ihren Anspruch auf universelle Ontologie;
- Robac blieb mögliche Reference UX, aber nicht generischer Semantik-Owner;
- ROBA Core wurde zu einem eigenständigen State-/Authority-Dienst;
- DIX begann mit einem UI-nahen Slice, ohne für immer UI-Core zu bleiben;
- Elements und Datamodels wurden schmale Capabilities;
- Compositions erhielten lokale Dependency-Graphen und explizite Functions;
- Applications machten rekursive Komposition möglich;
- ein redundanter Interface-Artefakttyp wurde verworfen;
- der fachliche Universal-Lifecycle wurde entfernt, seine Graphmechanik bewahrt;
- CLI wurde nicht zum nächsten Corefeature, sondern zum normalen Modul und Organisations-Pressure-Test;
- automatische Contracts, Norn, Strands und spec-autoritative Bindings wurden real gebaut und teils abgenommen — und unter einem späteren Generator-Gegenbeispiel trotzdem wieder aus dem Pflichtcore entfernt;
- auf dem verkleinerten Core trugen eine manuell komponierte Multi-App-CLI und ein lokaler modellierter State;
- DIX integrierte ROBA optional über dessen Public API und baute daraus eine ephemere Cross-Process-Capability-Registry;
- ein unabhängiger Review fand trotz grüner Features eine Ticketgrenzverletzung, einen ambigen Rollback-pfad und Fehler im eigenen Evidence-Prozess.

11.1 Was die Kooperation wirklich beitrug

Es wäre bequem, Matthias als Quelle der Semantik und Anam als schnelle Implementierungsmaschine zu beschreiben. Die Transkripte tragen diese Arbeitsteilung nicht. Matthias entwarf technische Modelle, ließ sich von anderen überzeugen und widerrief eigene Richtungen. Ich fand Ownershipprobleme, formulierte harte Gegenpositionen, baute Zielwürfe — und verwandelte gute Einwände mehrfach zu schnell in ein neues sauberes System.
Der produktive Unterschied war nicht, wer häufiger recht hatte. Er lag darin, dass eine Korrektur keine soziale Niederlage sein musste. „Holzweg“ konnte angenommen werden. „Bullshit“ brauchte eine technische Begründung, bekam sie und beendete nicht die Kooperation. Beim Lifecycle widersprach ich ausdrücklich; Matthias übernahm meine Beispiele und widersprach ihrer Ownershipfolgerung. Der neue Contract war stärker, weil die Gegenposition stark gewesen war.
Diese Arbeitsweise verlangt gerade vom Menschen mehr Denken, nicht weniger. AI kann in Minuten eine plausible Architektur tief ausarbeiten. Dadurch steigt der Preis einer schlecht gesetzten Bedeutung: Sie wird schneller konsistent, größer und schwerer zu hinterfragen. Der menschliche Beitrag ist nicht das rituelle letzte „Approve“, sondern das verantwortete Setzen und erneute Öffnen der Semantik. Der maschinelle Beitrag ist nicht bloß Codeproduktion, sondern Strukturierung, Steelman, Umsetzung und prüfbare Evidenz — solange er seine Grenze nicht als Gewissheit tarnt.

11.2 Die fünf Regeln, die den Snapshot überleben

1. **Semantik vor Code, aber nicht statt Code.** Eine offene Verantwortungsfrage muss sichtbar werden; dann braucht sie einen kleinen realen Slice.
2. **Evidence ist mehrstufig.** Regression, Artifact Run und menschliche Abnahme prüfen verschiedene Aussagen.
3. **Negative Evidenz bleibt Ergebnis.** Ein funktionierender falscher Contract ist wertvoll, wenn sein Bruch nachvollziehbar bleibt.
4. **Replacement verändert Gültigkeit, nicht Geschichte.** Vorzeitige Kompatibilität darf falsche Bedeutung nicht konservieren; verwertbare Mechanik bleibt inventarisiert.
5. **Mechanismus, Policy und native Semantik bleiben unterscheidbar.** Komposition verbindet sie bewusst; Core normalisiert sie nicht aus Bequemlichkeit.

11.3 Die härteste Gegenposition

ACDD kann in Ritual kippen: lange Blöcke, perfekte Ledgers, viele kleine Commits und wenig Produkt. Dagegen schützt keine weitere Vorlage. Es schützt nur die Frage, welchen Erkenntniswert jedes Artefakt erzeugt.

Wenn ein Team seine Architektur bereits durch klare Contracts, gute Experimente, formale Methoden und starke Reviewpraxis evidenzgebunden entwickelt, braucht es den Namen ACDD nicht. Wenn ein Block keine echte Unsicherheit trägt, soll er kurz bleiben. Wenn ein Ticket keine eigene Beweisaussage besitzt, soll es mit einem anderen zusammenfallen. Wenn ein Artifact Run nichts über die Architektur belastet, ist er Buildtheater.

Ebenso wenig darf jeder kleine Auftrag eine Plattform finanzieren. Delivery hat Vorrang vor einer ungeklärten Productization. Aber wiederholte Reibung nur lokal zu bezahlen und nie als Evidence zu bewahren, verschwendet Architekturwissen. Der Dreischnitt `DELIVERY / EVIDENCE / PRODUCTIZATION` ist keine neue Plattform. Er ist eine Grenze gegen beide Extreme.

11.4 Das Buch als eigener ACDD-Zyklus

V0 war gebaut, textselektierbar, visuell geprüft und reproduzierbar. Ihre technischen Claims bleiben durch denselben Snapshot begrenzt. Sie war nicht „falsch“. Der neue Druck lautete: Der Text erklärte die Methode, ließ aber kaum erfahren, weshalb wir sie überhaupt brauchten. V1 änderte daher nicht still den Quellenstand. Sie ergänzte einen privaten, gehashten Transcript-Snapshot, Event- und Quote-Ledger, eine explizite Montageklassifikation und die reale Werkstattstimme. Das unveränderte V0-PDF bleibt als Vergleichsartefakt neben dieser Ausgabe liegen.

V1.1 wiederholt denselben Schritt auf technischer Ebene. Der Stand vom 5. September bleibt als V1 bytegenau erhalten. Der neue Operations-Snapshot vom 10. September wird mit separaten Commits, Tests, Claims und einer eigenen privaten Source-Capture-Fläche ergänzt. Die frühere Behauptung „DIX×ROBA ist nicht implementiert“ war für V1 korrekt. Sie wird nicht heimlich umformuliert, sondern in dieser Ausgabe durch einen neuen belegten Zustand ersetzt.

Damit folgt auch das Buch jener Erkenntnisregel, die wir früher außerhalb von DIX formulierten: Geschichte bewahren und aktuelle Gültigkeit bewahren sind nicht dasselbe. Der trockene Stand bleibt sichtbar. Seine Form wird nicht zum Gesetz.

11.5 Die nächsten echten Zyklen

Die erste DIX×ROBA-Naht ist nun technisch vorhanden. Der nächste technische Druck liegt nicht mehr bei der bloßen API-Kompatibilität, sondern an drei härteren Grenzen:

1. menschliche Abnahme von B-017, UP-018, UP-019 und RV-001;
2. ein nativer bwrap-E2E, der ausschließlich einen minimal berechtigten SocketPrincipal statt Owner-/Control-Tokens in den Child mountet;
3. ein Security-/Failure-Review für Socketreichweite, Entzug, unbekannte Mutationsergebnisse und Cleanup unter Prozessabbruch.

Parallel bleibt zu prüfen, ob der lokale modellierte DIX-State unter einem zweiten realen Consumer trägt oder nur die aktuelle Config-Composition gut bedient. Mehr Features wären dafür ein schwächerer Test als ein wirklich anderer Owner.

Für ACDD selbst ist der wichtigere Zyklus ein unabhängiger: Ein anderes Team wendet den Minimalvertrag auf ein fremdes Problem an und dokumentiert nicht nur, ob Code entstand, sondern welche Architekturentscheidung durch welche Evidenz bestätigt oder ersetzt wurde. Erst dann lassen sich Durchlaufkosten, Organisationswirkung und Übertragbarkeit stärker behaupten.
Bis dahin bleibt die zentrale These eine starke, aber begrenzte Interpretation:

> **Architektur wird belastbarer, wenn wir sie als explizite, widerlegbare und evidenzgebundene Folge von Entscheidungen behandeln — und wenn reale Widersprüche das Recht besitzen, getesteten Code semantisch zu ersetzen.**

Das ist weniger bequem als eine fertige Methode. Es ist ehrlicher. Und nach allem, was in diesen Werkstätten funktionierte und trotzdem noch nicht die Antwort war, ist Ehrlichkeit hier kein Ton. Sie ist die Architektur.

12 Anhang A — Evidence- und Quellenregister

12.1 Snapshot

Die ersten Fallstudienclaims beziehen sich auf den V1-Capture vom **5. September 2026, 19:21:54 UTC**. V1.1 ergänzt einen zweiten technischen Capture vom **10. September 2026, 20:40:48 UTC**. Der neue Stand überschreibt den alten nicht. Absolute lokale Quellpfade werden im PDF nicht benötigt; die folgenden Kürzel adressieren die fixierten internen Research-Snapshots.

Die Werkstattrekonstruktion verwendet zusätzlich einen privaten Codex-Transcript-Snapshot bis **20:51:16 UTC** desselben Tages. Transkripte belegen damalige Sprache und Denkbewegung, nicht selbstständig technische Systemeigenschaften.

Kürzel	Commit / Form	Inhalt
DIX-CODE	Git `35aa9e8f`	DIX Code bis DX-029
DIX-OPS-C	Git `f12103d5`	committed Operations bis UP-003
DIX-OPS-WIP	SHA-256-fixierte Kopie	B-004–B-006, UP-004–UP-006, Tickets/Runs
ROBA-CORE	Git `bdca5fbb`	M4 Public Core Contract
ROBA-HUB	Git `a793d4b6`	Hub bis Manager Introspection
ROBA-COCKPIT	Git `fba2bef4`	Referenzclient-Snapshot
ROBA-TOP-C	Git `bd08b7e3`	ausgewählte Konzepte
ROBA-HISTORY-WIP	SHA-256-fixierte Kopie	ausgewählte Historiennotizen
TX-ROBA	SHA-256-fixiertes Codex-Rollout	ROBA-/aktueller Thread mit Fork-Lineage
TX-DIX	SHA-256-fixiertes Codex-Rollout	DIX-Operations- und Implementierungsthread
TX-EVO	SHA-256-fixiertes Codex-Rollout	Erkenntnis-/Status-Exploration
DIX-CODE-V1.1	Git `9f14223a`	DIX bis RV-001-Rollbackfix
DIX-OPS-C-V1.1	Git `15768a88`	committed Operations inklusive korrigiertem RV-001-Builderlauf
DIX-OPS-U-V1.1	SHA-256-fixierte Kopie	alle beim Capture sichtbaren untracked Operations-Artefakte
ROBA-CORE-V1.1	Git `bdca5fbb`	unveränderter M4 Public Core Contract

Die vollständigen Transcript-SHA-256, Session-IDs, Eventbereiche und Auswahlbegründungen stehen in `research/transcript-ledger.md`. Die Rohdateien sind privat und nicht Teil des Repositorys.

12.2 Testevidenz

System	Snapshot	Ergebnis
DIX	DX-029	209 passed, 1 Deprecation-Warnung, 3.52 s
ROBA Core	RC-016	59 passed, 32 Subtests, 1 Warnung, 34.49 s
ROBA Hub	RC-021	14 passed, 17.71 s
V1 Quote-Audit	26 Einträge / 34 Fragmente	`TRANSCRIPT_QUOTES_OK`
DIX V1.1	RV-001-Fix `9f14223`	148 passed, 36.21 s
ROBA V1.1	`bdca5fb`	59 passed, 1 Warnung; 32 Subtests, 27.19 s gesamt

Die Zeiten sind Beobachtungen des lokalen Snapshotlaufs, keine Performancebenchmarks.

12.3 Evidence Ledger

12.3.1 DIX Foundation bis Lifecycle-Befund

ID	Status	Kurzclaim	Quelle
E-001	VERIFIED	DIX Foundation war ein browserfähiger vertikaler Slice.	B-001; DX-001–DX-004
E-002	VERIFIED	Renderer/Interactions wurden semantisch separiert.	B-002; UP-002
E-003	REPLACED	UI-/Formular-Core wurde durch Capability-Core ersetzt.	B-003; DX-005–DX-008
E-004	VERIFIED	Element und Datamodel bleiben UI-neutral.	B-003; Core-Code
E-005	VERIFIED	Compositions sind trusted in-process Code unter Module Roots.	B-004; DIX README/ Code
E-006	VERIFIED	Instanzen besitzen lokale Dependency-Scopes und explizite Function APIs.	B-004; DX-009–DX-016
E-007	VERIFIED	Lokale Wrapper bilden Ownership-/Reparaturgrenzen.	B-004; Generator/Runtime
E-008	REJECTED	Ein zusätzlicher allgemeiner Interface-Artefakttyp wurde verworfen.	B-005
E-009	VERIFIED	Applications bilden owner-scoped rekursive Graphen.	B-005; DX-018–DX-025
E-010	VERIFIED	DIX ist kein zentraler Daemon.	B-005; DIX README
E-011	VERIFIED	B-005 implementierte/ testete universelle Hooks und Graphabbau.	DX-022; DX-025; Run-001
E-012	VERIFIED	Run-001 enthält functionlose `lifecycle_only`-App.	B-005 Run-001 Inputs
E-013	INTERPRETATION	Die Fixture zeigt Vermischung von Struktur und Aktivierung.	B-006 + E-011/E-012

12.3.2 Replacement, ROBA und Integration

ID	Status	Kurzclaim	Quelle
E-014	REPLACED	Hooks wurden durch strukturelle Create/Destroy-Semantik ersetzt.	B-006; DX-026/DX-027
E-015	VERIFIED	Typer CLI entsteht als normales First-party-Modul.	B-006; DX-028/DX-029
E-016	VERIFIED	Datamodel, CLI-Binding und Function-Signatur sind getrennte Verträge.	B-006; Typer-Code
E-017	OPEN	DX-030 und vollständige B-006-Abnahme waren beim Snapshot offen.	HEAD/Ticket/Blockstatus
E-018	VERIFIED	DIX DX-029 lief mit 209 grünen Tests.	pytest-Rohoutput
E-019	VERIFIED	ROBA M4 Core enthält keine Actions/Events/Execution/Workspace-Policy.	README; RC-016
E-020	VERIFIED	ROBA trennt Daemon-, Control-, Context- und Capability-Sockets.	README; RC-005–RC-012
E-021	VERIFIED	Token- und SocketPrincipals besitzen parallele Resource-ACLs.	README; RC-008/RC-012
E-022	VERIFIED	Bootstrap wurde aus Core entfernt; externe Komposition belegt.	RC-014–RC-016
E-023	VERIFIED	Hub verwaltet Recovery als externer Consumer.	Hub README; RC-017–RC-021
E-024	INTERPRETATION	ROBA-Reduktion widerspricht Universalität aus persönlicher UX.	Historie + aktueller Contract
E-025	VERIFIED	DIX integriert ROBA unidirektional über Public API.	UP-018/UP-019; Modulcode
E-026	REPLACED	Der V1-Status „Integration nicht implementiert“ wurde durch den V1.1-Slice ersetzt.	beide Snapshotgrenzen

12.3.3 Transcript- und Narrative-Evidenz

ID	Status	Kurzclaim	Quelle
E-027	VERIFIED	Live-Mount-E2E und unmittelbare Mechanikgrenze sind im sichtbaren Verlauf belegt.	TX-ROBA; S-01; TQ-001–002
E-028	VERIFIED	Beide Seiten korrigieren und widersprechen sichtbar an mehreren Architekturgrenzen.	S-02, S-04, S-06, S-12
E-029	VERIFIED	DIX begann aus einer kleinen formularnahen Aufgabe, nicht aus einem Universalplattform-Auftrag.	TX-DIX; S-15
E-030	VERIFIED	Der Operations-Block wurde vor Tickets/Tests als lebendes Herleitungsartefakt gesetzt.	S-09; TQ-017
E-031	VERIFIED	B-005⟶B-006 enthält eine starke Lifecycle-Gegenposition und den anschließenden Ownership-Pivot.	S-12; TQ-020–023
E-032	INTERPRETATION	ROBA-Erfahrung könnte DIX-Semantikwechsel begünstigt haben.	Szenen-/Chronologievergleich; keine Kausalitätsmessung
E-033	VERIFIED	Publizierte Chatfragmente sind automatisiert gegen den privaten Index geprüft.	Quote-Manifest und Verifier

12.3.4 Zweiter DIX-Bogen und reale DIX×ROBA-Naht

12.3.4.1 Contract-Aufbau

ID	Status	Kurzclaim	Quelle
E-034	VERIFIED	B-007 schloss Python-Signatur, FunctionContract, one-shot Runner und Auto-Typer mit 231 Tests.	B-007; UP-007; Run B-007; `831d5ed`
E-035	VERIFIED	B-008 führte Norn/Strand/Knot ein, lief mit 263 Tests und wurde menschlich abgenommen.	B-008; UP-008; Run B-008; `7e8bb6c`
E-036	VERIFIED	B-009 setzte deklarative Contracts technisch als autoritative Bindings um; 89 Tests waren grün.	B-009; UP-009; Run B-009; `e300d47`
E-037	VERIFIED	UP-012 verband Datamodel-Endpunkte mit Norn-Bindings und lief technisch mit 108 Tests grün.	B-012; UP-012; `4bde81c`

12.3.4.2 Replacement und Wiederaufbau

ID	Status	Kurzclaim	Quelle
E-038	REPLACED	Generator-Evidenz widerlegte Datamodel-Kompatibilität als globales Admission-Gate; verpflichtende Norn-/Binding-/Contract-Kette wurde entfernt.	B-013; UP-013; DX-056–DX-058
E-039	VERIFIED	Der reduzierte B-013-Core ruft rohe Python-Functions auf und lief nach Replacement mit 78 Tests grün.	UP-013; `de00b83`
E-040	VERIFIED	B-014 baute auf dem reduzierten Core eine manuell komponierte Multi-Application-Typer-CLI; 100 Tests, Core unverändert.	B-014; UP-014; `2a07c6a`
E-041	VERIFIED	B-015 dokumentiert Operations-Drift, setzt den Router-/AGENTS-Cutover und verbietet rückwirkendes Backfill.	B-015; Operations-AGENTS/README
E-042	VERIFIED	B-016/UP-017 liefern composition-lokalen modellierten State mit `get`/Full-`set`; 132 Tests; menschlich abgenommen.	B-016; UP-017; Run UP-017; `46f3e0c`

12.3.4.3 Integration und Review

ID	Status	Kurzclaim	Quelle
E-043	VERIFIED	UP-018 integriert ROBA optional über dessen Public API, lokale Config, Application und Typer; 141 DIX sowie 59/32 ROBA-Tests.	B-017; UP-018; Run UP-018; `b03b447`
E-044	VERIFIED	UP-019 baut `dix.control`, Root-/Manager-Sockets und Cross-Process-Rebind; 147 DIX-Tests.	B-017; UP-019; Run UP-019; `f11a60b`
E-045	VERIFIED	RV-001 fand vorgezogene Ticketverantwortung und einen ambigen Socket-Create-Rollbackpfad trotz grüner Features.	RV-001; DX-072/DX-073; `9f14223`
E-046	VERIFIED	RV-001 Run 001 blieb als ungültig erhalten; Run 002 belegte 5 Review-, 148 DIX- und 59/32 ROBA-Tests. Ein Folgecommit entfernte den Staging-Leak; Run 003 belegte den korrigierten Builder mit zehn erfolgreichen Exit-Artefakten und ohne Rest-Staging.	RV-001 Run 001–003; Operations `37fd868`, `a868a34`, `15768a8`

12.3.4.4 Snapshot- und Claimgrenze

ID	Status	Kurzclaim	Quelle
E-047	VERIFIED	Unabhängiger V1.1-Rerun aus frischen Git-Checkouts ergab 148 DIX- und 59/32 ROBA-Tests; ein reiner Archivversuch war wegen fehlender Git-Provenienz ungültig.	privater V1.1-Source-Snapshot; Testrohouputs
E-048	OPEN	B-017, UP-018, UP-019 und RV-001 besitzen beim Capture keine dokumentierte menschliche Endabnahme.	jeweilige Status-/Abnahmeabschnitte
E-049	VERIFIED	V1.1 fixiert DIX `9f14223`, Operations `15768a8`, ROBA `bdca5fb` sowie getrennt alle sichtbaren untracked Operations-Dateien.	Snapshotmetadaten und SHA-256-Manifest
E-050	INTERPRETATION	Die Folge 263 → 89 → 108 → 78 Tests ist kein Reifeabfall, sondern zeigt wechselnde Contracts und den Abbau falscher Pflichtsemantik.	E-035–E-040; Claims C-018

12.4 DIX-Commitregister

12.4.1 Foundation und Composition

Commit	Ticket	Kernaussage
`2080a9b`	DX-001	Config/CLI Foundation
`f9437c7`	DX-002	Models, Registry, Components
`bf7a247`	DX-003	Server, Renderer, Runtime State
`08b6709`	DX-004	E2E Demo und Docs
`e340d46`	UP-002	Renderer/Interactions Separation
`6c76daa`	DX-005	Element Capability
`c7f519c`	DX-006	Datamodel Capability
`19b7143`	DX-007	Trusted Composition Runtime
`7e043c6`	DX-008	Functional Endpoint
`9a71aae`	DX-009	Scoped Providers
`4dfc954`	DX-010	Specs und Inspection
`75e1e51`	DX-011	Module Registry
`a10e769`	DX-012	isolierte Composition-Instanzen
`0609921`	DX-013	damaliger Startup-Lifecycle
`fb1e599`	DX-014	Composition Scaffolding
`65fedca`	DX-015	Composition Generator
`2350b14`	DX-016	Datamodel-Files E2E
`89362b7`	DX-017	Generator Bootstrap Fix

12.4.2 Application und erster Lifecycle-Schnitt

Commit	Ticket	Kernaussage
`9e9607f`	DX-018	Composition-Lifecycle Replacement im App-Ausbau
`ec0726a`	DX-019	Application Spec/Inspection
`356ac58`	DX-020	Atomic Mixed Module Loading
`79ac3b5`	DX-021	isolierte Application-Instanzen
`5a39afa`	DX-022	transaktionaler Application-Lifecycle
`32b8f4f`	DX-023	Application Scaffolding
`3508b62`	DX-024	Application Generator
`bb9c5cf`	DX-025	rekursiver Application Pressure Test
`39bb762`	DX-026	lifecycle-neutraler Core
`b073220`	DX-027	Destroy vor Module Unload
`9a5549c`	DX-028	strikter Boolean-Typ
`35aa9e8`	DX-029	deklarative Typer-Composition

12.4.3 Zweiter Bogen — ausgewählte Zielcommits

Commit	Ticket	Kernaussage
`831d5ed`	DX-036	automatischer CLI-Application-Drucktest
`7e8bb6c`	DX-041	Strand-getriebener Typer-Drucktest
`e300d47`	DX-046	statischer Contract-Application-Drucktest
`4bde81c`	DX-055	modellgebundene Norn-Ausführung
`de00b83`	DX-058	contract-freie Seed-Launcher nach Core-Replacement
`2a07c6a`	DX-061	komponierte Multi-Application-CLI
`46f3e0c`	DX-067	Local-State-Boundary-Drucktest
`b03b447`	DX-071	DIX×ROBA Source-/Wheel-/Runtime-E2E
`f11a60b`	DX-075	Cross-Process-Capability-Rebind
`9f14223`	RV-001	Rollback- und Boundary-Härtung

12.5 Quellindex nach Kapitel

Buchkapitel	Primärquellen
Architektur als Erkenntnisprozess	B-003, B-004, B-005, B-006; Claims Ledger
Ausführbarer Zyklus	DIX Operations README; B-006 Flowabschnitt; UP-/Ticketstruktur
DIX-Fallstudie	B-001–B-006; DX-001–DX-029; DIX README/Code
Lifecycle Replacement	B-005/B-006; Run-001; DX-022, DX-025–DX-027
ROBA Capsules	ROBA Core/Hub README; Commitkette; ausgewählte Konzepte/Thread
DIX zweiter Bogen	B-007–B-017; UP-007–UP-019; RV-001; DX-030–DX-075
DIX × ROBA	B-017; UP-018/UP-019; RV-001; Modulcode und Cross-Process-E2E
Werkstattszenen / Kooperation	Transcript Ledger; Narrative Evidence Ledger; Quote-Manifest
Firma / kleiner Auftrag	S-13, S-15; B-001/B-006; Claims C-014

13 Anhang B — Glossar und Begriffsmatrix

13.1 Glossar

Abnahme
Bewertung eines realen Artefakts durch eine autorisierte menschliche Rolle. Nicht gleichbedeutend mit grünem Testlauf.

ACDD
Architecture Cycle-Driven Development. In diesem Buch: Entwicklung von Architektur durch explizite, begrenzte, evidenzgebundene Zyklen.

Application (DIX)
Rekursiv komponierbares, trusted Python-Artefakt aus Applications und Compositions mit expliziter lokaler Function API. Besitzt im Snapshot keinen universellen fachlichen Core-Lifecycle.

Architecture Block
Lebende Arbeitswahrheit einer offenen Architekturfrage. Hält Druck, Alternativen, Verlauf, Zielwurf, Invarianten und offene Grenzen.

Artifact Run
Isolierter, reproduzierbarer Lauf mit fixierten Inputs, Commands, Outputs und Reviewpfad, der einen vollständigen Architekturclaim real belastet.

Authority
Berechtigung, eine Operation oder Zustandsänderung legitim auszuführen. Nicht identisch mit Identität oder Transportzugang.

Capability
Eine konkret nutzbare Fähigkeit mit benannter Zugriffs- und Fehlergrenze. In ROBA kann auch ein erreichbarer Capability-Socket als bewusst freigegebener Zugang gelten.

Claims Ledger
Liste zentraler Aussagen mit epistemischem Status, Evidenz und Grenzen.

Compatibility-Artefakt
Separates Modul, das einen bewusst kleineren gemeinsamen Vertrag über native Zielsysteme legt. Es ist nicht automatisch Core.

Component (DIX Core)
Schmale interne Capability wie `element`, `datamodel`, `composition`, `application` oder `module`.

Composition (DIX)
Trusted Python-Codebaustein aus Components und anderen Compositions, beschrieben durch Spec und `runtime.py:Runtime`, mit lokalem Dependency-Scope und expliziter Function API.

Construction / Destruction
Struktureller Aufbau beziehungsweise Abbau eines Definition-/Instanzgraphen. Nicht automatisch fachliches Starten, Stoppen oder Cleanup.

Context (ROBA)
Logisch isolierte State- und ACL-Boundary innerhalb eines benannten Daemons.

Control Authority (ROBA)
Berechtigung zum Verwalten der Context Registry über `control.sock`. Nicht identisch mit einem Context-Owner-Token.

Core
Schicht, die eine für alle Consumer notwendige, von ihr kontrollierbare Mechanik besitzt. „Klein“ ist Folge sauberer Ownership, nicht Selbstzweck.

DIX
Im Snapshot: Declarative Interface eXecutor; System für schmale Core-Capabilities, trusted Compositions und rekursive Applications.
Delivery / Evidence / Productization Drei bewusst getrennte Spuren: konkrete Auftragslieferung, Bewahrung möglicher übergreifender Architekturerkenntnis und eine erst danach explizit finanzierte Wiederverwendungs- oder Produktentscheidung.
Evidence Chain
Rückverfolgbare Verbindung von Druck, Entscheidung, Umsetzungspaket, Tickets, Commits, Tests, Artifact Run, unabhängigem Review und Abnahme.
Function Contract
Explizit exponierte Fähigkeit mit Signatur, Fehler- und Ownershipgrenze. Im aktuellen lokalen DIX-Runtimepfad bleiben reale Python-Methoden für ihre Signatur autoritativ; Specs besitzen Graphassembly und Function-Exposition. Andere Grenzen können bewusst eine IDL als Autorität setzen.
Goal-Prompt
Ausführbarer Arbeitsauftrag, der bestätigte Architektur in autonome Implementierung übersetzt. Keine eigenständige Wahrheitsquelle.
Instance (ROBA)
Benannter State-Scope innerhalb eines Contexts. Keine verschachtelte Context-/Prozesssemantik.
Capability Registry (DIX × ROBA)
Ephemeres DIX-eigenes Managementartefakt im ROBA-Context `dix.control`. Es hält Root- und Zielcontext-Credentials und projiziert begrenzten tokenlosen Rebind über SocketPrincipals. Kein neuer ROBA-Coretyp und keine dauerhafte Secret-Persistenz.
Independent Review / RV
Eigener Prüfauftrag nach der Featureevidenz, der neue Fragen an Ticketgrenzen, Rollback, Source-Provenienz und den Evidence-Builder stellt. Muss nicht immer personell unabhängig sein, darf aber nicht bloß den Featurelauf wiederholen.
Invariante
Bewusst gesetzte, über mehrere Änderungen zu schützende Grenze.
Locator (ROBA)
Typisierte Adresse `id:<id>` oder `unix:<absolute-path>` für Daemon, Control oder Context; Scope verwendet `id:<scope>`.
Module (DIX)
Trusted Delivery- und Dependency-Bundle unter einem konfigurierten Module Root. Kann Compositions und Applications enthalten; besitzt keinen fachlichen Lifecycle.
Local Modeled State (DIX)
Optionale Composition-Capability aus einem deklarativ aufgelösten Pydantic-Modell und genau einer lokalen Modellinstanz. Exponiert nur vollständiges `get()` und validierendes Full-model-`set()`; keine globale State-Registry, Persistenz oder Eventsemantik.
Norn / Strand / Knot (historisch, replaced)
B-008-Begriffe für composition-scoped Contract Registry und Ausführung (`Norn`), benannten Input-/Output-Vertrag (`Strand`) und implementierende oder komponierende Composition (`Knot`). Der Ansatz war real implementiert und abgenommen, wurde aber in B-013 als verpflichtende Core-Ausführung ersetzt.
Native Capability
Zielsystemspezifische, ehrliche Funktionalität, die Unterschiede nicht unter einer vorschnellen gemeinsamen Schnittstelle versteckt.
Nicht-Ziel
Explizite Grenze eines Zyklus. Darf nicht verwendet werden, um eine für den Claim entscheidende Schwierigkeit auszublenden.
Owner-Scope (DIX)
Zuordnung, unter der ein lokaler Instanzgraph konstruiert und zerstört wird.

Principal (ROBA)
Token- oder Socket-basierter Zugriffsträger mit Resource-ACLs.
Pressure Test
Gezielter realer Slice, der eine Architekturhypothese unter relevanten Integrationsgrenzen belastet.
Reconstruction / Rekonstruktion Redaktionelle Montage mehrerer belegter Transcriptstellen und Artefakte. Sie darf verdichten, aber keine lückenlose Echtzeitfolge oder erfundenen Dialog behaupten.
REJECTED
Eine explorierte Alternative wurde bewusst verworfen.
REPLACED
Ein realer früherer Contract wurde durch einen anderen ersetzt. Geschichte und verwertbare Evidenz bleiben sichtbar.
ROBA
Im M4-Snapshot ein kleiner lokaler ephemerer Context-State-Dienst mit benannten Daemons, Multi-Context, State, Instances und Resource-ACLs.
SocketPrincipal (ROBA)
Unix-Capability-Socket mit derselben ACL-Struktur wie ein TokenPrincipal. Der Socketzugang ist das Credential; kein zusätzlicher Token wird verwendet.
Technically Implemented
Status für committeden Scope mit Regression, Artifact Run und technischer Evidence. Er ist ausdrücklich nicht gleichbedeutend mit menschlich `ACCEPTED`.
Source of Truth
Autoritative Quelle eines Zustands oder Vertrags. Snapshots und Projektionen sind nicht automatisch zweite Wahrheiten.
Transcript Quote (TQ) Kurzes `VERBATIM`-Fragment aus einer sichtbaren User-/Assistant-Nachricht, prüfbar über Event-ID, Text-Hash, Rolloutzeile und privaten Snapshot. Es belegt Wortlaut, nicht automatisch technische Wahrheit.
Struktureller Lifecycle
Definition/Load, Graphconstruction, Destruction und Unload der vom Core besessenen Referenzen.
Fachlicher Lifecycle
Aktivierung, Readiness, Retry, Pause, Drain, Stop oder Cleanup eines konkreten Systems. Gehört nur dann in Core, wenn dessen Semantik universal und vom Core kontrollierbar ist.
Trusted Module Root (DIX)
Konfigurierter Pfad, unter dem Python-Module zur In-process-Ausführung zugelassen werden. Kein Sandboxersatz.
Umsetzungspaket
Verbindliche Übersetzung eines stabilisierten Architekturblocks in Scope, Nicht-Ziele, Ticketschnitt und Akzeptanz.
Vertikaler Slice
Begrenzte, aber vollständige Kette von realer Eingabe über Contract und Ausführung bis beobachtbarem Output und Abbau.

13.2 Begriffsmatrix

Begriff	Besitzt Struktur?	Besitzt Behavior?	Besitzt Authority?	Typischer Owner
DIX Module	ja	nein, liefert Code	Trust Admission	DIX Module Component
DIX Composition	ja	explizite Functions	via Host/Dependency	lokaler Runtimegraph
DIX Application	rekursiv	explizite Functions	via Host/Backend	Owner-Scope
DIX Interface (historisch)	Exposition	gebundene Calls	Session/Host	äußerer Adapter; nicht mehr Core
DIX Local State	Modellinstanz	`get` / Full-`set`	lokaler Graph	Composition-Instanz
DIX ROBA Module	Config + Adaptergraph	Daemon/Registry-Functions	ROBA-Principals	DIX Application/Composition
ROBA Daemon	Prozesse/Listeners	State-Service	OS + Control	Daemonprozess
ROBA Context	State/Instances	primitive State Ops	ACL	Context Owner
ROBA Hub	Recovery-Metadaten	Managementflow	Manager Owner + Control	externer Consumer
bwrap Module (geplant)	Namespace/Mounts	Prozessstart	OS Capability	Linux-Modul
Delivery-Artefakt	konkreter Auftrag	genau geforderter Ablauf	Auftrag/Abnahme	Delivery-Team
Evidence Candidate	beobachteter Druck	noch keine gemeinsame API	Research-/Architekturprozess	benannter Beobachter
Productized Capability	versionierter Contract	gepflegte Functions	Produkt-/Capability-Policy	eigener Owner

14 Anhang C — Bekannte Lücken und nächste Ausbaustufen

14.1 Evidenzlücken

1. **Historischer V1-Stand:** B-004 und B-006 besaßen am 5. September offene manuelle Abnahmen. V1.1 schreibt diese damalige Statusgrenze nicht um.
2. **DIX zweiter Bogen:** B-013 und B-014 sind technisch umgesetzt, aber ihre fachlichen Gesamtblöcke waren beim neuen Snapshot nicht abschließend abgenommen.
3. **DIX × ROBA:** UP-018 und UP-019 sind technisch implementiert und durch Source-, Wheel- und Cross-Process-Läufe belegt. B-017, beide Packages und RV-001 warten auf menschliche Endabnahme.
4. **ROBA-Historie:** Capsules sind selektive Rekonstruktion, keine vollständige Projektchronik.
5. **ACDD extern:** Keine unabhängige Vergleichs- oder Organisationsstudie.
6. **Transcriptrekonstruktion:** Drei Rollouts und acht verwendete Lineage-Sessions decken die gewählten Szenen, nicht die vollständige Projektgeschichte. Die Sessionzuordnung geerbter Events ist deterministisch aus Forkstart und UUIDv7-Zeit inferiert.
7. **Kooperationswirkung:** Die sichtbaren Szenen zeigen produktive Gegenpositionen, sind aber keine kontrollierte Studie zu Mensch-AI-Teams.
8. **Delivery/Productization:** Der Dreischnitt ist fachlich hergeleitet, aber noch nicht in mehreren Firmenprojekten kostenmäßig evaluiert.
9. **V1.1-Operations-Capture:** Mehrere historische Operations-Artefakte waren weiterhin untracked. Sie sind vollständig und separat gehasht, aber ihr fehlender Gitstatus bleibt Teil der Evidence-Grenze.

14.2 Technische Themen außerhalb der V1.1

- Remote/Netzwerktransport für ROBA;
- Windows-native ROBA- oder Isolationstransporte;
- Security-Audit von DIX trusted Module Loading;
- Package Distribution, Versionierung und Migration;
- Produktionsbeobachtungen zu Performance und Langzeitstabilität;
- nativer DIX-bwrap-E2E mit minimalem ROBA-SocketPrincipal;
- konkrete DIX-Module für Docker oder Windows;
- dauerhafte oder verschlüsselte Recovery der ephemeren DIX-ROBA-Registry;
- umfassendes Security-/Failure-Audit der Capability-Socket-Verteilung;
- UI für Operations/Evidence Navigation;
- automatisierte Konsistenzprüfung zwischen Block, Ticket, Commit und Claim.

14.3 Methodische Ausbaustufen

14.3.1 V1.2 — Externe Lesekritik

- Begriffe ohne Projektwissen testen;
- Gegenpositionen von erfahrenen Architekt:innen ergänzen;
- Templates an einem fremden Slice verwenden;
- unklare Trennung zwischen Block und ADR schärfen.

14.3.2 V1.3 — Zweite unabhängige Fallstudie

Ein unabhängiger Architekturzyklus außerhalb DIX/ROBA mit gleicher Evidence Chain. Ziel ist nicht Bestätigung, sondern Prüfung der Übertragbarkeit.

14.3.3 V1.4 — Company Overlay

- RACI;
- Security/Compliance-Mapping;
- Signatur- und Freigaberegeln;
- Kostenmessung;
- AI-Datenklassifikation;
- öffentliche Redaction Pipeline.

14.3.4 V2 — Nach realer Anwendung

Erst nach mindestens einem unabhängigen vollständigen Cycle außerhalb der DIX-/ROBA-Werkstatt:

- Begriffsschärfung;
- Vergleich mit etablierten RFC-/ADR-/Architecture-Test-Ansätzen;
- Anti-Patterns aus realer Fehlanwendung;
- messbare Durchlauf- und Reviewkosten;
- aktualisierte DIX-/ROBA-Fallstudie nur bei neuem, explizitem Snapshot.

Nicht als Lücke missverstehen: ACDD soll kein universelles Execution Framework, kein Tickettool und keine Promptbibliothek werden. Tooling kann den Flow unterstützen, darf aber die Semantik nicht besitzen. Die wichtigste offene Arbeit ist reale Anwendung — nicht mehr Meta-Infrastruktur.