WORKSHOP BOOK · VERSION 1.1

Architecture Cycle-Driven Development

Architecture as an evidence-bound process of discovery, told from the DIX and ROBA workshop

Real workshop scenes, wrong turns put to the test, strong counterarguments, and deliberate replacements — verifiably reconstructed from DIX and ROBA.

Matthias Fulz · developed collaboratively with Anam

V1.1 · Sources as of September 10, 2026

Technical status: DIX RV-001 · Operations B-017 · ROBA RC-016 · Transcripts: 8 sessions

Snapshot and reading status. This V1.1 adds the technical status as of September 10, 2026, 20:40 UTC, without overwriting the V1 cutoff of September 5. The supplementary transcript cutoff remains unchanged at September 5, 20:51 UTC. `VERIFIED`, `INFERENCE`, `INTERPRETATION`, `OPEN`, `REJECTED`, and `REPLACED` remain distinct. This work is a narrative, textbook, and case study — it makes no claim of product maturity or effectiveness.

English-edition note. This is a direct translation of the German V1.1 manuscript, not a separately sourced rewrite. Transcript excerpts identified by `TQ-*` were originally written in German. Their English wording is a translation; the quote ID continues to identify the verified German source.

About this edition

This edition is typeset as a verifiable workshop book. Real scenes carry the domain-level explanation without becoming raw chat. Short original excerpts are verified through event and quote IDs; technical claims remain tied to code, commits, tests, and artifact runs. Evidence IDs lead to the register, and the build remains reproducible.

Edition	V1.1 · September 10, 2026
DIX	`9f14223` · RV-001 · 148 tests green
Operations	`15768a8` · B-017/RV-001 · Some human acceptances pending
ROBA Core	`bdca5fb` · RC-016
Transcripts	3 rollouts · 8 lineage sessions used · private SHA-256 snapshot
Status	Workshop book/case study; no assurance of product maturity or effectiveness

> **Reading rule:** Where the sources support only a domain-level reading, the text says `INTERPRETATION` or `INFERENCE`. B-017, UP-018, UP-019, RV-001, and other open DIX states that have not yet received human acceptance remain explicitly `OPEN`, even when their technical evidence is green.

Contents

Preface: The Green Run

On September 5, 2026, `run-001/` held exactly the kind of artifact we had long been looking for. No screenshot of a demo, no loosely jotted-down command, no "should work." The folder contained inputs, generated code, inspection outputs, a complete runtime smoke test, an artifact tree, a verification sequence, and a script that could rebuild the run in isolation. The application graph was loaded, created, started, invoked, and destroyed again. No instances remained afterward. The tests were green.

That was a good moment. Not ironically good, and not merely good "for a prototype." B-005 had turned individual DIX Compositions into a recursive application graph. Dependencies ran in local scopes. Two root instances could exist separately. Error paths and teardown were tested. A review could follow actual files.

Then, among the input artifacts, an Application stood out that did not expose a single domain-level Function. It was called `lifecycle_only`. The Core could still start it, set it to `active`, and stop it again. That was exactly what the fixture had been built for: to prove that the lifecycle works even without a Function API.

It proved it. And that was where the problem began.

> **MATTHIAS · ARCHITECTURE SPACE** TQ-020 · GERMAN ORIGINAL: VERBATIM, abridged · ENGLISH TRANSLATION
>
> "[…] under no circumstances just mindlessly nod this through now […]." It was "an extreme edge case" — precisely the point where easy agreement would have been worthless.

My response at the time was not a yes. I argued against it at length and for good technical reasons.

> **ANAM · COUNTERPOSITION AT THE TIME** TQ-021 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "I would not couple lifecycle to exposed functions."

A worker can operate without a DIX Function. An event consumer, file watcher, scheduler, or HTTP service can have active behavior even though its actual interface lies outside the Function API. If only apps with exposed Functions received a lifecycle, the result would be meaningless dummy Functions or, worse still, behavior would move into imports and constructors. The counterposition was not stupid. It was technically consistent and strong enough to be taken seriously.

Matthias did not counter it by denying the examples. He accepted them and moved the ownership question one level up.

> **MATTHIAS · SHIFT IN THINKING** TQ-022 · GERMAN ORIGINAL: VERBATIM, excerpt · ENGLISH TRANSLATION
>
> "it needs lifecycle for the reasons you give and for exactly the same reasons it must not be there"

The systems needed lifecycle. That did not imply that a general-purpose composition core was entitled to own *the* lifecycle. Workers, HTTP services, CLIs, and externally managed processes needed different activation, readiness, retry, drain, and cleanup contracts. The Core could neither know this meaning universally nor control it completely.

My next response drew from this the sentence that underpinned B-006:

> **ANAM · LATER DISTILLATION** TQ-023 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "The Core knows instance lifetime, but no domain-level lifecycle."

A few hours later, the reserved `start`/`stop` semantics had been removed. Graph construction, local scopes, explicit destroy, inspection, and a large part of the tests remained. The earlier work was not rewritten as an embarrassing mistake. It had built a technically sound mechanism while also producing the evidence that allowed its overly broad meaning to be replaced.

This book is about precisely such moments.

No Textbook Alongside the Story

The first edition of this book was substantively sound and deliberately dry. It explained Architecture Cycle-Driven Development, or **ACDD**, through DIX and ROBA. It pinned commits, tests, artifact runs, pending acceptances, and claims. What it barely conveyed was the actual workshop: why a wrong direction was plausible at the moment it emerged; why we were honestly pleased with intermediate results; how often a new, clean model was merely the next form of the same conflation; and why disagreement between human and machine became a condition of work rather than a disruption.

V1 does not attach this story to the textbook as a personal appendix. The story carries the technical knowledge. Terminal evidence can lead into a question from that time; a strong counterposition into code; green code into a semantic rupture; a replacement into a general review rule. Readers who only want to check the facts will still find Evidence IDs, commit states, and claim boundaries. Those reading from beginning to end should also discover how these facts acquired their meaning.

ACDD denotes neither a certification nor a finished framework. It is a way of working: architecture is developed as a sequence of explicit, bounded cycles of inquiry tied to evidence. A target design becomes concrete enough for code. The code is checked automatically and used as a coherent artifact. The findings may then confirm, sharpen, or replace the architecture.

Two Systems, No Retrospective Heroic Tale

The first workshop is **ROBA**. There, bwrap, capability reduction, and dynamic mount propagation worked surprisingly well early on. What failed to stabilize for a long time was the working model derived from them. Profiles, Workspaces, Actions, Runners, Clients, Child Daemons, and Contexts produced ever more plausible boundaries. Several times, we considered one of them the decisive one. Several times, a small edge case showed that we had merely made a personal way of working configurable and then mistaken that for openness.

The second workshop is **DIX**, "Declarative Interface eXecutor" at the time of the snapshot. DIX began with a small task centered on forms and moved through six closely documented cycles toward narrow Core Capabilities, trusted Compositions, and recursive Applications. In just a few days, its development was faster than many ROBA rounds. This is no proof of a miracle and no certain causal account. A fair interpretation says only this: the negative evidence accumulated in ROBA may have helped us recognize certain traps in DIX more quickly. The code and operations snapshot supports the DIX facts; the causal claim remains limited.

The roles are not smoothed over either. Matthias is not the infallible architect dictating truth to a machine. He let functional models convince him, pushed them further, and later broke them open again. I am neither an oracle nor that handy idiot who supposedly just needs strict enough prompting. I structured large amounts of material, formulated contracts, implemented target designs, and contributed counterpositions. But I also built a new, clean model too quickly whenever a good objection appeared. Our cooperation did not work through harmony. It worked because saying "wrong track" or "bullshit," or calling a hard stop, did not end the collaboration, and neither side had to save face.

> **AUTHORIAL CONTRACT · V1** TQ-026 · GERMAN ORIGINAL: VERBATIM, excerpts · ENGLISH TRANSLATION
>
> The editorial brief contained both closeness and a boundary: "you soulless tin contraption :P" — and: "please really just don't hallucinate now."

The first permits this edition a voice of its own. The second binds it.

Source Contract

Workshop scenes come from three privately frozen Codex rollouts with eight lineage sessions actually used. Short original fragments are verified through TQ IDs against event ID, timestamp, text hash, rollout line, and the SHA-256 of the raw transcript. Longer scenes are treated as reconstruction or paraphrase. We claim no hidden AI thought processes. An inner monologue appears only where the movement of thought actually exists in writing. Private paths, credentials, and company context are not published.

Technical claims remain tied to two fixed snapshots. The first ends on **September 5, 2026, 19:21 UTC**:

- DIX code through DX-029, commit `35aa9e8`, 209 reproduced tests;

- ROBA Core RC-016, commit `bdca5fb`;
- ROBA Hub RC-021, commit `a793d4b`;
- operations WIP hashed separately;
- DIX DX-030 and full B-006 acceptance still `OPEN`.

V1.1 adds the state as of **September 10, 2026, 20:40 UTC**:

- DIX through the RV-001 fix, commit `9f14223`, 148 reproduced tests;
- committed operations through `15768a8` and all visible untracked operations files hashed separately;
- ROBA Core unchanged at `bdca5fb`, 59 tests and 32 subtests;
- B-017, UP-018, UP-019, and RV-001 supported by technical evidence, but human acceptance still `OPEN`.

V0 and V1 remain preserved as byte-exact comparison artifacts. The new state does not overwrite the old one. The transcripts provide evidence of the language and shifts in the problem at the time. For kernel behavior, API contracts, and test counts, code, artifacts, and raw outputs remain authoritative. Source references **E-…** lead to the Evidence Ledger; **TQ-…** to the versioned quote manifest.

How to Read the Status Words

> **VERIFIED** — directly supported by code, a commit, a test, an artifact run, or unchanged visible wording.

> **INFERENCE** — a strong conclusion drawn from multiple pieces of evidence, but not directly stated in an artifact.

> **INTERPRETATION** — a present-day substantive interpretation that remains open to debate.

> **REJECTED** — an explored direction was deliberately discarded.

> **REPLACED** — an earlier, real contract received a new validity status; its history and usable mechanisms remain preserved.

> **OPEN** — not decided, not implemented, not accepted, or not reliably reconstructable.

The Warning About Our Own Method

ACDD can itself become theater: long blocks, perfect ledgers, many small commits, and no delivered result. An architecture block can degenerate into a diary; evidence into decoration; AI into a very fast machine for an unresolved direction. No additional template protects against this.

Every artifact must serve an epistemic purpose: make uncertainty visible, bound a contract, focus code, prove behavior, or enable acceptance. Anything that serves none of these purposes has no place in the process on principle alone.

The strongest statement in this book is therefore not “adopt our folders.” It is:

> **Make visible what you currently believe you know. Build only enough to put that assumption under real pressure. And allow the findings to change your architecture.**

1 The ROBA Workshop: When the Mechanics Already Worked

1.1 11 p.m.: An Honest Technical Success

The first finding in this story is not a failure. On the contrary: on July 13, a fairly demanding Linux chain worked. A parent operator ran in a bubblewrap environment with exactly one effective capability, `CAP_SYS_ADMIN`. It owned a propagation island that was both `shared` and `slave`. Beneath it ran an already-started child process without Linux capabilities and with `NoNewPrivs:1`. The operator added recursive bind mounts after the child had started and removed them again. The child saw both changes. One target remained read-only; another wrote back to the underlying source. In the child, that same island was only `private,slave`: propagation in, no equivalent feedback out.

EVIDENCE · TERMINAL TQ-001 · VERBATIM, paths anonymized

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

This was no claim made in a diagram. `findmnt` showed the mounts actually in effect, the write attempt hit the RO boundary, and the child process wrote back through the RW target. The mechanical hypothesis had been put under load and confirmed.

MATTHIAS · WORKSHOP, JULY 13 TQ-002 · GERMAN ORIGINAL: VERBATIM, abridged · ENGLISH TRANSLATION

"[…] mounts work -> we'd checked that before but now again, solidly"

The sentence contains both relief and impatience. This was not our first encounter with the mechanism, but now we had really nailed it down as a complete E2E. My reaction at the time followed logically: Codex should become the first real consumer. No generic profile system up front; first a concrete instance, then extract what is shared from two real cases.

That was reasonable. Yet it already contained the next danger. Once a mechanism works, our minds start giving it a working model. Mounts become sandboxes, sandboxes become profiles, profiles become workspaces, and workspaces become an organizing scheme for all tools. The transitions seem small. In the end, a kernel mechanism has quietly decided how people should work.

1.1.1 Interim Assessment 1 — A Mechanism Is Not Yet Policy

The test supported exactly these claims:

- A privileged operator can provide dynamic RO/RW mounts.
- An unprivileged child process can consume them.
- The mount option remains visible as an effective write boundary.
- The chosen propagation can carry changes into a running namespace without giving the child `CAP_SYS_ADMIN`.

It did not support:

- that "sandbox" is the right universal interface;
- that a workspace corresponds to a mount group;
- that tools need the same paths or profiles;
- that Linux mount semantics can meaningfully be normalized across Windows, containers, or cloud storage.

LESSON — A capability can be technically complete and still be only raw material at the domain level.

1.2 Visible, Readable, Writable: Three Different Truths

That same night, we stumbled over a smaller but revealing false assumption. A Codex process was running with an additional write sandbox. Its home was visible and read-only in the outer bwrap. I was too quick to treat that as though the inner sandbox also protected read visibility of a credential file located there. Matthias had the process check the boundary without printing the contents.

> **MATTHIAS · CORRECTION** TQ-004 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "especially since you're on the wrong track right now anyway "

The file was visible, the same UID could read it, and only writing was blocked by the outer RO mount. My statement had conflated two boundaries: an additional policy for write access and the read capability already granted by the outer namespace.

> **ANAM · RESPONSE AT THE TIME** TQ-005 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "Yep, you're right — I was on the wrong track."

The important technical rule is not "read-only is insecure." It is: **Visibility, readability, and writability must be proven separately.** A credential does not become invisible because its directory was mounted read-only. Anyone who needs real read isolation must remove the file from the namespace, use a separate credential process, or establish another explicit boundary.

The next day, the question shifted from security to runtime truth. We had tried to reproduce host paths in the sandbox as identically as possible, create materialized homes, and at the same time overlay a specific writable CWD. This worked in part, and precisely for that reason produced stacks of mounts that were hard to read. A subdirectory could exist in the materialized home but have no effect at the location that the running namespace actually saw. `$HOME` could be read-only while a CWD mount overlaid at a deeper path remained effectively writable.

> **MATTHIAS · SHIFT IN THINKING, JULY 14** TQ-003 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "my biggest problem right now isn't that there's a difference from the host here but that I don't know what's actually active through all the materialized files scattered around somewhere and coming together"

The sentence changed the design criterion. The goal was not maximum similarity to the host, but inspectable runtime truth. A deliberately differently named working mount could be more honest than a perfect mirror if `mountinfo`, the launcher, and state paths clearly showed where every active file came from.

This was especially relevant to Codex: session resume depended on the effective CWD, not on the host project we had in mind. Paths ceased to be cosmetic representations as soon as persistence used them as identity.

1.2.1 Interim Assessment 2 — Paths Become a Contract

```
Profile source     says: what is statically intended
Runtime state      says: what a specific instance has changed
Launcher           says: how the two were assembled
/proc/mountinfo    says: what the process actually sees
effective CWD      says: what session identity and resume depend on
```

None of these artifacts may silently speak for another. A DIX module may later model Linux paths and mounts explicitly. DIX Core itself should universalize neither Unix CWD nor host path identity.

1.3 The Long Loop Was Not "Too Little Configuration"

After this point, ROBA did not simply shrink in a straight line. There was a real personal need: separate private and professional contexts, project-specific chats, dedicated auth and runtime areas, tmux as the daily UX, isolated dev installations, global and local actions, sandboxes with dynamic capabilities. These needs were not invented. They merely made the models dangerously convincing.

We tried profiles, workspaces, domains, sessions, and hierarchies. A central context file was meant to describe what applied within an area. Scripts and actions were meant to map personal concepts onto transferable mechan-

ics. Robac was meant to make everything visible and pleasant to use. Runners were meant to be interchangeable; later, that very interchangeability was recognized as an additional source of uncertainty. Events, `call`, `emit`, child daemons, package mappings, and workspace specs came within reach or were already in code.
Every individual move had a good reason. A central context layer promised less scattered configuration. Actions promised reuse. Child daemons promised isolation. Robac promised the practical experience that an abstract core lacked. The problem was not that we were adding random nonsense. The problem was that every plausible solution embedded the still-unresolved fundamental meaning a little deeper in the system.
On July 20, this led to a drastic cut.

> **MATTHIAS · ARCHITECTURE SPACE** TQ-006 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "let's first clean up core ruthlessly"

Events, Exec, and interchangeable runners were to disappear. Context state, instances, and a small public API were to remain. I objected at one point: an additional new `ClientApi` alongside several existing layers would merely relocate the conflation. We consolidated into transport, Context API, and a high-level client. Shortly afterward came:

> **MATTHIAS · CERTAINTY AT THE TIME** TQ-007 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "the three apis would then be the final target"

Today, this sentence is valuable precisely because "final" did not stay final. It shows no naivety. It shows that an architectural state must be genuinely binding at the moment of the decision so that it can be built. But ACDD must not turn that into eternal truth. Commitment to the next design and irreversibility are different things.
Commit `3a817d…` rebuilt ROBA as a focused core. The later M4 boundary RC-016 made its exclusions explicit:

```
no action system
no event system
no executor
no sandbox semantics
no workspace or tool ontology
no child-context hierarchy
no UX policy
```

What remained was not deliberately useless minimalism, but an independent capability: ephemeral context state with instances, principals, ACLs, and a clear separation of transport and authority. [E-019]

1.4 The Sentence That Exposed False Universality

Three days after the tough core discussion, the deeper diagnosis was out in the open. We had repeatedly tried to parameterize individual preferences until the result looked like universal freedom. A `sandbox.start` action was configurable. A workspace profile could choose different mounts. A user context could set personal concepts. Yet the world on offer remained a world of sandboxes, workspaces, and context profiles.

> **MATTHIAS · ARCHITECTURE SPACE, JULY 23** TQ-008 · GERMAN ORIGINAL: VERBATIM, two fragments · ENGLISH TRANSLATION
>
> "we have everything we need" — and in the same train of thought, the correction: it was "just a normalization of the semantics for MY WAY!!!!"

Capital letters are not a method. The thought behind them is: another person does not have to understand their work as a system-wide sandbox hierarchy. Perhaps they use a sandbox only in CI, no tmux, no domains, and no persistent AI sessions. The fact that our model has many options does not make it neutral.
My distillation at the time captured the mistake unusually well:

> **ANAM · ASSESSMENT AT THE TIME** TQ-009 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "We tried to standardize **your concrete realization of freedom** as a universal freedom interface."

The alternative was not to abandon every shared standard. The same dialogue distinguished a **concrete shared work process** from all conceivable technical realizations. A team can, for example, jointly define architecture

phases, invariants, references, and review status. Whether someone carries out that process with AI, bwrap, CI, a text editor, or a sheet of paper belongs to a different layer.

> **ANAM · ASSESSMENT AT THE TIME** TQ-009 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "We standardize the **concrete shared work process**, not every conceivable way of realizing it technically."

Here lies an early, but still unfinished, precursor to ACDD. It would be wrong to claim in retrospect that the complete cycle had already been settled that day. All the evidence establishes is this: the process/realization boundary was formulated explicitly after the sandbox had broken down as a universal interface.

1.4.1 Interim Assessment 3 — Customizable Is Not Open

A useful test for apparently open systems is:

> Can a second user meaningfully use the promised capability without adopting the first user's ontology?

If not, the solution may be excellent personal policy. That is legitimate. It simply must not present itself as a universal core contract.

1.5 A Clean Seed, a Second ROBA, and an Immediate Break

In early August, we finally wanted to work vertically: launch Robac, display actions, select a workbench area, configure bwrap, and open Codex and zsh in tmux. I formulated a detailed seed from UX, profiles, action paths, and context values. It was orderly. That was precisely where its danger lay.

Matthias applied a small pressure test: the same context should be usable by two different ROBA installations. The seed had already pulled paths, runtime projection, and UX together. The second consumer was therefore not a variation, but a contradiction.

> **MATTHIAS · REVIEW** TQ-010 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "your seed is already — sorry for saying it directly :P — bullshit to me"

The word was rough; the reasoning was precise. A context intended to carry shared truth suddenly contained the local resource projection of exactly one ROBA instance. We had conflated context truth, runtime, UX, and path layout. More includes or overrides would not have repaired that ownership.

Shortly afterward, we took the opposite direction: "Robac first" was meant to bring real UX in early as a pressure test. That, too, was reasonable. But while designing a session action, Robac quietly became the controller again of what was actually supposed to be independently consumable.

> **MATTHIAS · SHIFT IN THINKING** TQ-011 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "through robac first we've already dismantled the actual goal again "

> **ANAM · BOUNDARY AT THE TIME** TQ-012 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "Robac is not the session's controller, but just one possible consumer that has been started."

This boundary remained valuable. It does not mean UX should not drive architecture. A reference client is often the best place to expose unusable APIs. It simply must not silently own generic mechanics. The later `roba-ctl-hub` was therefore built as a separate consumer: recovery and management functionality outside the core, with no Robac dependency. [E-023]

1.6 What Actually Remained in the Core After the Removals

The M4 core separates four levels of access and authority:

```
daemons/<daemon-id>/
├── daemon.sock       process status and stop; OS capability
├── control.sock      Context Registry/Gateway; control token
└── contexts/<id>/
    ├── ctx.sock      direct context access; TokenPrincipal
```

```
    └── sockets/<id>.sock
                    dynamic SocketPrincipal
```

A **Daemon** is a process, failure, mount, and trust boundary. A **Context** is a state and ACL boundary. An **Instance** is a named state scope within that context, not a hidden process or child-context hierarchy. Typed locators `id:…` and `unix:…` prevent bare IDs and relative socket paths from meaning different things depending on the caller. [E-020]

This separation answers three different questions:

1. May an operational participant see or stop the daemon process?
2. May a control authority manage and address contexts?
3. May a principal read, write, manage, or grant further rights to specific context or instance state?

OS file permissions on `daemon.sock` do not automatically grant access to content. A control token is not an owner token for every context. A direct `ctx.sock` bypasses only locator resolution, not ACLs.

1.6.1 SocketPrincipals: Capability Without a Secret Environment

For short-lived or sandboxed tools, tokens in environments, files, or command lines are awkward and easily exposed by accident. ROBA therefore added SocketPrincipals. A dedicated Unix socket has the same resource ACL structure as a TokenPrincipal. The reachable socket is the credential; an additional token on this access path is an error.

ACLs distinguish the resources `context` and `instance:<id>` and the rights `read`, `write`, `manage`, `grant`. Changes take effect on the next request; deletion removes the listener. A bwrap process can have exactly this socket mounted into it without carrying a secret in its environment. [E-021]

The counterposition belongs here too: Unix socket capabilities are not platform-neutral. A Windows target needs a native alternative. It would be wrong to hide the different security properties beneath a vague “portable socket.”

> **PRINCIPLE** — Native capability first. A compatibility contract emerges only when a real consumer needs precisely its smaller shared guarantee.

1.6.2 Why Bootstrap Also Had to Move Out Again

An empty context initially seemed impractical. Declarative bootstrap files were meant to build instances, ACLs, and state during creation. But as soon as the core interprets such files, it owns format, merge, conflict, include, templating, persistence, and rollback semantics. Higher-level personal composition was returning through a supposedly small convenience function.

RC-014 removed bootstrap. RC-015 proved that an external consumer can build the same state through individual public operations. The core deliberately promises no transaction across multiple calls: earlier successful calls remain visible if a later one fails. RC-016 fixed this smaller, more honest contract in place. [E-022]

The hub showed that “outside” does not mean “unstructured.” It could own the manager context, local anchor, and recovery metadata as a concrete higher layer. The core API remained strong enough without imposing the concrete hub model on every user.

1.7 From Shared Tool State to Explicit Behavior

In August, the exploration fell into a familiar pattern again. If Vim, VS Code, Sway, tmux, and other tools are to cooperate, a shared state model seems natural: Editor, Buffer, Window, Workspace, Active Item. One tool projects its state into it; others read it.

Even “Editor” breaks down. Vim buffers, VS Code tabs, files, remote documents, and processes do not naturally share one ontology. A tool without tabs would have to emulate tabs or simulate `unsupported` as an empty value. Multiple truths also emerge: native tool state, normalized copy, adapter state, update time, and conflict ordering.

> **MATTHIAS · ARCHITECTURE SPACE, AUGUST 13** TQ-013 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> “we kept coming back to state thinking in the sense of "data"”

The thinking moved toward actions, functions, and finally closures: not all tools should have to hold the same state; an artifact should express what behavior it actually offers. But overcorrection was possible here, too. “Everything is a function” would merely have hidden values and prompted a universal execution engine.

> **ANAM · COUNTERPOSITION** TQ-014 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> “Values are not the problem. Shared, prematurely universalized data models are the problem.”

This boundary remains important to this day:

- Values describe legitimate state.
- Functions expose deliberate behavior.
- ROBA state is useful when it is truly authoritative and shared.
- A projection of arbitrary tool internals does not become authoritative simply because it is stored centrally.
- DIX compositions and applications can own the consumable function contract without imposing the same ontology on every tool.

1.8 When DIX Appeared, ROBA Was Not Suddenly “Solved”

On September 4, a young codebase existed outside ROBA, born of a small task closely related to forms. By that point, DIX already had explicit function APIs, local composition scopes, recursive application graphs, and a module model. The comparison with ROBA seemed strikingly apt:

```
DIX
= Delivery + local dependency graphs + Function Contracts

ROBA
= optional external authoritative state + Authority
```

The temptation would have been to immediately rebuild both and “marry” them. The earlier work argued against precisely that. ROBA had a public API; DIX could later consume it like any other capability. There was no reason to make ROBA depend on DIX or invent new core semantics before the first integration.
The clean hypothesis is unidirectional:

```
DIX Application/Composition
        ↓ normal public API call
ROBA Context / Instance / Principal
```

ROBA remains the source of truth for the state stored there and its ACL. DIX remains the owner of the exposed behavior contract. A bwrap module can later mount a SocketPrincipal without “sandbox” becoming the ontology of both systems. [E-025]

> **HISTORICAL STATUS BOUNDARY** — In the V1 snapshot, this DIX×ROBA integration was not implemented. V1.1 now provides evidence of the optional public API slice and an ephemeral capability registry; their human B-017/package acceptance remains open. [E-026, E-043–E-048]

1.9 What Remained of the Long Workshop

ROBA is not a failed preliminary stage of DIX. That narrative would be convenient and wrong. The mount mechanics work. The path and capability tests remain valuable. Today, the context daemon has a narrower, technically independent responsibility. TokenPrincipals and SocketPrincipals, multi-context, instances, and ACLs are not merely scars from a wrong turn.
It would be equally wrong to romanticize the loops. Several times, we stayed too long at the level of possible overall systems. We kept building on unclear concepts. We declared intermediate states the final boundary too early. Detailed answers occasionally made a direction more convincing than its ownership basis justified. Real UX arrived too late or, as soon as it arrived, immediately became a candidate for universal semantics again.
The usable finding is more nuanced:

1. A working mechanism quickly creates excessive semantic confidence.
2. Configurable personal policy remains personal policy.
3. A reference client should exert pressure, not own generic mechanics.
4. Core reduction is progress only if external composition remains genuinely possible afterward.

5. Authoritative shared state is valuable; universal mirroring of tool state is not automatically valuable.
6. A later composition layer can take on removed higher-level functionality in the right place without retrospectively treating the smaller core as a mistake.

The workshop therefore does not end with “at last we had the solution.” It ends with a better working contract for the next question: we must show what meaning we are establishing, what real finding can support it, and which layer may actually own it. That is exactly where ACDD begins.

2 Architecture as a Process of Inquiry

ROBA did not first give us a method. It gave us a sequence of contradictions that were hard to explain away: mounts could be correct while the working model remained open; a runtime could be configurable while its semantics remained personal; a client could provide the best UX and thereby become precisely the wrong owner of generic mechanisms. ACDD begins with the question of how such contradictions become reliable architectural knowledge before memory or the next clean design smooths them over, rather than with a new file structure.

2.1 Architecture Begins Before the Module Boundary

Software architecture is often described through structures: components, layers, ports, events, data flows, and deployment boundaries. These structures are important, but they are not the beginning. Before them lies a question of meaning: **What responsibility should the system own in the first place?**

A `start()` hook can be cleanly typed, perfectly tested, and correctly integrated into a dependency graph. Yet it remains open whether "starting" is a universal property of the Core or domain-level behavior of a particular Application. A shared state model for editors can be elegantly normalized. Yet it remains open whether buffers, tabs, and files really constitute the shared semantics of all editors — or merely the conceptual model of the first tool integrated.

Architecture is therefore not primarily the decision about *how* something is coupled. It is the explicit decision about *what* may be coupled, who owns the meaning, and what claim an interface actually makes.

> **INVARIANT** — A technically clean abstraction can be semantically wrong. Good types, tests, and layers reduce implementation errors; they do not automatically legitimize the chosen responsibility.

2.1.1 Four Levels of an Architectural Statement

For ACDD, a strict separation helps:

1. **Observation:** What is directly visible in the current system or use case?
2. **Assumption:** What explanation or expectation are we using that has not yet been proven?
3. **Decision:** What contract are we setting for the next bounded iteration?
4. **Invariant:** What boundary should remain stable across multiple iterations?

Example:

- Observation: A test Application without functions executes a `start` hook when created.
- Assumption: Every Application has a meaningful operational lifecycle.
- Decision: The Core calls `start` and `stop` in a defined graph order.
- possible invariant: Graph teardown must be deterministic and complete.

The later pressure test can refute the assumption without destroying the invariant. That is exactly what happened in DIX: the universal lifecycle was removed, while the knowledge about graph order, rollback, scopes, and deterministic teardown remained usable. [E-011–E-014]

> **ASSUMPTION** — "Every Application has a meaningful operational lifecycle" was not an observed fact, but the generalization to be put under pressure. Labeling it as such is precisely what makes a later replacement possible without rewriting history.

2.2 Architecture as a Testable Hypothesis

An architectural decision is almost always a hypothesis about future use. "Compositions need their own dependency scope" means more than that we create objects locally today. It claims that local isolation is more viable for parallel instances with different configurations than a global registry state. "Code remains authoritative for signatures" claims that a second copy of the signature in TOML produces more drift than benefit.

A useful architectural hypothesis has five properties:

Property	Question
explicit	Can we state what assumption we are making?
bounded	To which slice and which non-goals does it apply?
executable	Can a concrete artifact run put it under pressure?
refutable	What finding would force a correction?
traceable	Are decision, code, test, and acceptance connected?

"We are building a flexible plugin system" meets none of these conditions. It specifies neither trust boundary nor ownership, instantiation, signature source, or error semantics. B-004 only became sound when "plugin" was broken down into concrete responsibilities: trusted module root, static spec, `runtime.py:Runtime`, local dependency scopes, explicit wrappers, and a Function API whose authority resides in code. [E-005–E-007]

2.2.1 Counterposition: Doesn't Architecture Need to Be Stable Up Front?

In regulated or safety-critical systems, certain boundaries must be established much more firmly before implementation: hazard controls, data protection boundaries, cryptographic protocols, or irreversible migration rules cannot carelessly be "tried out under pressure."

ACDD does not contradict this. It does not say that every decision must arise from production experiments. It demands that the source of certainty be named honestly. An external standard, a formal proof, a threat analysis, or a prototype can each be evidence. The cycle must be adapted to risk and reversibility. What ACDD rejects is an unsupported equation of a clean diagram with correct semantics.

2.3 Semantics Before Implementation

"Semantics before implementation" does not mean discussing things for months. It means: before code hides a decision and thereby makes it effectively binding, we articulate the fork in meaning.

A typical fork is:

```
Variant A: Core manages domain-level start/stop behavior.
Variant B: Core manages only structural graph lifetime;
           domain-level behavior is a normal Function.
```

Only once this fork is visible can an appropriate pressure test be built. Without it, tests would probably only verify that `start()` is called at the programmed point. They would not ask whether this turns an Application without functions into an operational unit.

Semantic work provides at least:

- the actors and their authority;
- the resources affected;
- the direction of dependency;
- the source of truth;
- the claim about errors and rollback;
- the explicit non-goals;
- a finding that could refute the decision.

2.4 Vertical Pressure Tests Instead of Hypothetical Extensibility

Broad architectural designs like to optimize for possible futures: multiple renderers, platforms, editors, sandboxes, auth backends, event systems, and remote targets. Each possibility produces additional types, adapters, and configurations. The system looks extensible before a single complete process works reliably.

A vertical pressure test takes a different slice:

```
real input
  -> real contract
  -> real composition
  -> real execution
  -> observable output
  -> complete teardown
```

It is deliberately narrow, but not artificial. B-001 brought config, interface, runtime state, API, and browser presentation together. B-003 brought Element, Datamodel, Composition, an explicit Function, and an HTTP endpoint together. B-005 built a recursive App/Composition graph, generated a consuming artifact, and destroyed the graph again. [E-001, E-003, E-009, E-011]
The value lies not in the number of integrated features. It lies in an architectural claim actually crossing the boundaries between multiple layers.

> **COUNTERPOSITION** — A slice that is too small can create false confidence. If persistence, concurrency, or trust is decisive for the core claim, the test must not set it aside as "later." Non-goals are not permission to avoid the hardest relevant boundary.

2.5 Why Green Code Is Not Enough

Automated tests answer questions that someone has previously translated into assertions. They find many errors, but they can answer the wrong question perfectly.
The B-005 system could correctly check:

- start order along the dependency graph;
- rollback on an error;
- stop order;
- isolation of parallel root graphs;
- complete teardown.

All these properties are technically valuable. None on its own answers whether `start` and `stop` should be reserved Core hooks at all. The artifact run made the missing question visible because a human considered the generated system as a unit of meaning rather than merely a set of assertions.
This leads to three complementary forms of verification:

1. **Regression:** Does the code behave as the current contract requires?
2. **Artifact run:** Does the slice work outside the test fixture as a coherent system?
3. **semantic acceptance:** Is the contract still the right one in actual use?

None replaces the others.

> **EVIDENCE** — For DIX DX-029, the snapshot reproduced 209 green tests. This provides evidence for the committed technical state against its assertions. It expressly does not prejudge the still-open DX-030/B-006 acceptance.

2.6 Replacement as a Legitimate Tool

Backward compatibility is valuable when external users rely on a legitimate contract. It is harmful when it preserves a responsibility recognized as wrong before such a contract has even been stabilized.
ACDD uses **Replacement** deliberately:

- The old semantics are named.
- The concrete contradiction is supported by evidence.
- The new boundary is formulated as a replacement, not as an additional option.
- Usable technical knowledge is identified.
- Compatibility is preserved only when a real consumer requires it.

A common mistake would be to offer both paths after B-006:

```
core_lifecycle = true | false
```

That would look flexible, but would stabilize two incompatible meanings in the Core. DIX instead chose the narrower contract: construction and destruction are structural Core operations; activation is normal, explicit Function semantics. [E-014]

> **REPLACED** — Replacement does not erase history. The earlier model remains visible as evidence, precisely so that the new contract does not appear to be a self-evident truth from the outset.

2.6.1 Preserve History, Change Validity

This status vocabulary did not emerge from code alone. In an earlier meta-exploration, I had praised a new formulation by saying it was "the first sentence we don't have to take back." Matthias objected not to the recognition, but to the courtroom framing behind it: defending, retracting, subtracting errors, salvaging a remainder.

> **MATTHIAS · SPACE OF INQUIRY** TQ-015 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "we don't neeed to want to take anything back we have to keep testing what we've learned against the previous state and expand or condense it ;)"

The wording carries the typos and pace of the original. Its substantive meaning is more precise: an earlier state was real under earlier evidence. New evidence may narrow its scope of validity, differentiate its terms, or replace its contract. It does not undo history.
My correction at the time added the necessary guardrail:

> **ANAM · ASSESSMENT AT THE TIME** TQ-016 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "We distinguish between **preserving history** and **preserving current validity**."

Without the first part, architectural history becomes a victor's narrative. Without the second, "evolution" becomes an excuse to perpetuate refuted claims. The status values `REJECTED` and `REPLACED` accomplish both: they preserve the path of inquiry while ending a contract's current validity.

2.7 Stability or Mere Habituation?

A system feels stable after some use. This can mean that its contract holds. It can also mean that everyone involved has learned to work around its breaks.
Signs of habituation rather than stability:

- Documentation regularly explains special cases that the contract supposedly unifies.
- Wrappers have to synchronize hidden states.
- new consumers need access to details intended to be internal;
- "flexibility" comes mainly from flags;
- tests reproduce only the implementation, not the user process;
- faulty behavior is rationalized as user error;
- no one can name a finding that would change the architecture.

Stability is stronger: multiple real slices use the same contract without hollowing out its meaning; open boundaries remain explicit; the next extension adds mechanism instead of retrofitting displaced policy into the Core.

2.8 Cooperation as a Testable Area of Architecture

Roles shifted in the cycles shown. Matthias often brought a strong prior understanding and far-removed edge cases. I could quickly organize large bodies of material, translate an objection into APIs, invariants, tickets, and tests, and carry the implementation through. At other moments, the decisive technical counterposition came from me, and the semantic rupture only through Matthias's renewed scrutiny. "The human thinks, the machine implements" would therefore be just as wrong as "AI designs, the human confirms."
The productive pattern was:

```
one side creates pressure
→ the other builds the strongest plausible form from it
→ both examine the contract, not the harmony
→ artifacts are allowed to contradict both
→ the authorized person takes responsibility for the decision
```

Human authority here is not a claim of human infallibility. It denotes responsibility: under the working contract shown, only the human can accept the domain-level target state, bear risks, and approve a change. The machine may disagree, gather evidence, and report technical completion. It must not simulate user acceptance.

2.8.1 The Alignment Reflex

Cooperative language carries a risk of its own. A response can begin with "Yes, exactly," structure the objection elegantly, and merely reinforce its plausibility in the process. Even adding "I'm not just saying yes" proves

no critical examination. What matters is not the tone, but whether at least one real counterposition has been formulated with costs, a contract, and a finding that would refute it.
Conversely, mechanical disagreement is no mark of quality. The lifecycle scene was strong because the counterposition had real grounds in workers, services, and error handling. Only against this strong version did it become visible that the technical necessity of lifecycle did not yet establish Core ownership.

> **COOPERATION INVARIANT** — Disagreement must be possible without loss of face; agreement must be legitimized by evidence, not by closeness. A blunt "wrong track" can be more substantively cooperative than politely perpetuating false semantics.

2.9 The Minimal Epistemic Contract

ACDD does not demand many documents for a cycle. It demands a minimal contract:

```
We observe X.
We assume Y.
For this iteration, we decide Z.
We exclude A, B, and C.
We implement exactly slice S.
We expect evidence E1…En.
Finding F would refute the decision.
```

Everything else — block format, ticket system, CI, goal prompt, reviewer roles — is a matter of putting it into practice. Without this contract, activity results. With it, architectural knowledge can emerge.

2.10 What ACDD Explicitly Is Not

Confusion	Why it does not hold	ACDD boundary
Big Design Up Front	The future is treated as fully modelable.	Fix only enough of the target architecture to make a relevant slice testable.
arbitrary trial and error	Experiments have no explicit hypothesis or evidence chain.	Name the pressure, assumption, refutation, and finding in advance.
"code first, document later"	Code silently turns open semantics into fact.	Make the real fork in meaning visible before the contract.
ticket bureaucracy	Activity and fragmentation replace knowledge.	Tickets only after a stable target design and for each provable increment.
retrospective architectural history	The final state erases plausible wrong directions.	Preserve Rejected/Replaced and negative evidence.
prompt standardization	Instruction is confused with an executable rule.	Build mature standards as modules, validators, generators, and tests.
AI architecture theater	Large amounts of text and diagrams simulate certainty.	Tie claims to code, tests, artifacts, and human acceptance.
perpetual backward compatibility	False foundational semantics survive as a second path.	Deliberately replace before stabilization; plan real external migration separately.

ACDD is thus neither purely planned ahead nor purely emergent. It sets a bounded contract so that implementation can deliberately produce evidence, and retains enough openness for that evidence to change the contract again.

3 The Executable Architecture Cycle

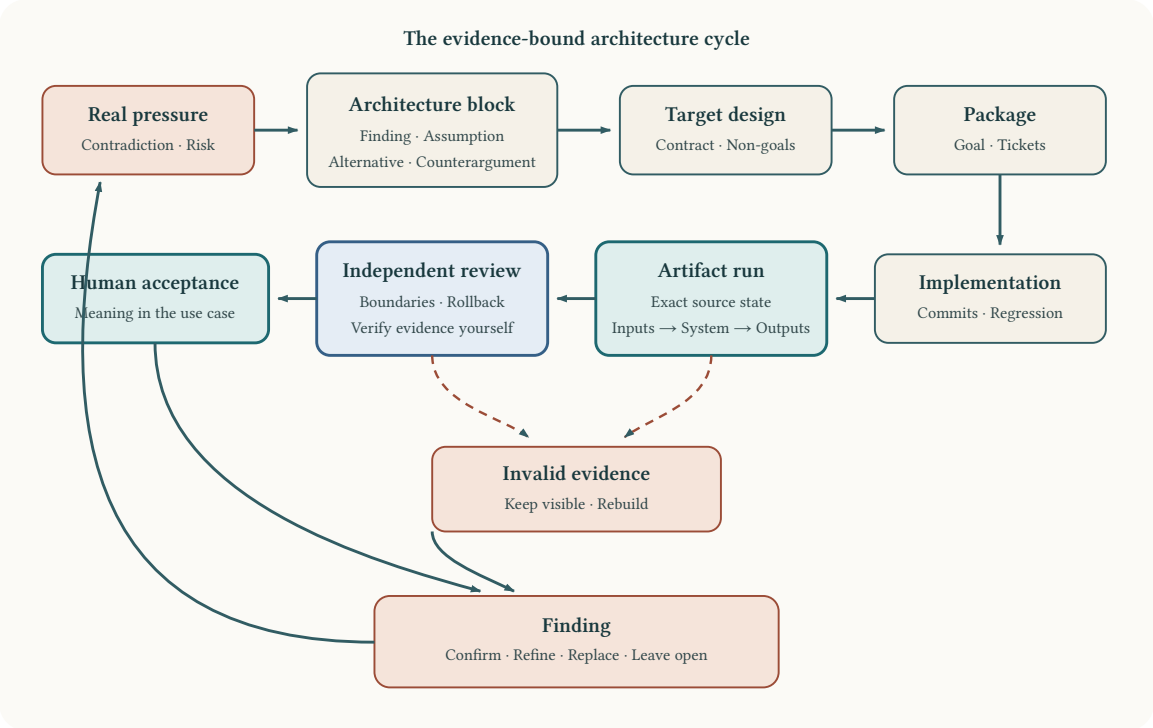

Figure 1: The ACDD cycle: pressure is converted into new evidence through explicit semantics, bounded implementation, and real acceptance.

3.1 A Chain of Evidence, Not a Waterfall

The ACDD cycle has a sequence, but no illusion of complete predictability. The architecture block precedes implementation because unresolved semantics must be visible. The artifact run and independent review precede acceptance because only a slice that exists and has been checked against its own blind spots can be assessed in practice. The next cycle can then replace the previous decision.

The operational chain is:

```
real pressure
→ living architecture block
→ cooperative refinement and counterposition
→ binding target design
→ implementation package / Execution Contract
→ small tickets and commits
→ regression
→ isolated artifact run
→ independent boundary/rollback review
→ manual acceptance
→ confirmation, refinement, or replacement
```

“Executable” does not mean that every step is automated. It means that the target design is translated into a concrete, verifiable change and that its evidence consists of more than opinion.

3.2 1. Real Pressure

A cycle begins with a contradiction, need, or risk, rather than a desired abstraction. Examples:

- The renderer owns template and interaction semantics that would be lost when switching representations.
- A composition needs two parallel instances with different configurations.

- An application with no functions is activated at the domain level by a reserved hook.
- A sandbox can receive mounts dynamically, but nobody can say what overarching workspace semantics should follow from them.

A good pressure statement describes observable behavior and avoids specifying the solution in advance:

> "Creating a structural app instance invokes behavior even though the app publishes no function."

Worse would be:

> "We need a more flexible lifecycle framework."

The second statement jumps straight into a solution category and can obscure the actual error — lifecycle in the Core.

3.3 2. Living Architecture Block

The **architecture block** is the current working truth for an open question. It is neither a specification nor a chat archive. It records:

- initial pressure and goal;
- terms and their current meanings;
- substantiated findings;
- alternatives and counterpositions;
- open decisions;
- rejected intermediate states;
- the course of significant corrections;
- the point at which an implementable target design becomes sufficiently stable.

In DIX, the block initially took the place of tickets and test cases. That mattered: while the meaning of "Interface," "Composition," "Application," or "Lifecycle" was still shifting, early tickets would merely have turned unstable semantics into work orders.

3.3.1 Workshop Scene: First the Block, Then Its Derivatives

On September 1, I had already proposed a tidy operations structure for the new DIX material: blockers, tickets, test cases, and a dedicated AI context. Organizationally, it looked orderly. Substantively, it was too early. Matthias corrected the direction, rather than individual filenames: the foundation block first needed to record what existed, which architecture questions were open, and how the target design was developing. Tickets, test cases, and the acceptance package could follow once the meaning held up.

> **ANAM · CORRECTION AT THE TIME** TQ-017 · GERMAN ORIGINAL: VERBATIM, excerpt · ENGLISH TRANSLATION
>
> "Block first as a living cooperation/derivation artifact"

The word *living* does not mean arbitrary. The block may include earlier intermediate states, but its current target design must be unambiguous. And the word *derivation* does not mean chat dump. The raw dialogue remains a source; the block condenses which observation created which fork and why a decision currently holds. Only then can tickets be small without losing meaning.

> **WARNING** — A block is no license for endless exploration. It must reduce the number of real forks. If it merely keeps collecting more possibilities, either real pressure or a decision is missing.

3.3.2 The Block Rule

For ongoing exploration:

1. The block is the primary truth.
2. Raw chat is a source, but not automatically a contract.
3. Decisions are recorded in the block in our own words.
4. Only a stable target design produces a package, tickets, and acceptance criteria.
5. After the run, the history remains visible; the final state is not retroactively written up as the only solution ever considered.

3.4 3. Cooperative Refinement and Active Counterposition

Cooperation is not consensus production. Its value lies in expanding the decision space and putting weak assumptions under pressure.

A useful counterposition asks, for example:

- Is the real problem a missing mechanism or personal policy?
- Is shared state being invented when only functions need to be shared?
- Is the abstraction native to the target systems or merely a compatibility layer?
- Who owns the source of truth?
- What happens with two parallel instances?
- What information would a consumer need to know that the contract says should be hidden?
- Would the proposed capability make sense even without the current client?
- What semantic assertion is hidden in a supposedly technical hook?

AI can compare large amounts of existing material and structure counterexamples here. The human remains responsible for deciding what meaning is intended. A model can recognize that two contracts collide; it cannot legitimize which organizational responsibility should apply.

3.5 4. Binding Technical Target Design

A target design is implementable when the following questions are settled:

- What public responsibility is added, changed, or removed?
- Which layer owns it?
- What inputs, outputs, and errors apply?
- What trust and authority boundary applies?
- Which invariants must tests check?
- Which non-goals prevent scope expansion?
- Is this an extension or a replacement?

The target design does not need to settle every future question. It must isolate the open questions that do not block the current slice.

3.5.1 Implementable Versus Merely Interesting

Exploration Only	Implementable Target Design
“Applications could have lifecycle.”	“Core constructs and destroys graphs but invokes no domain-level hooks.”
“CLI should be declarative.”	“A first-party Typer module binds only named options to explicitly exposed app functions.”
“ROBA could share state.”	“A DIX module uses only the public ROBA client API; ROBA Core has no knowledge of DIX.”
“Sandbox should be portable.”	“bwrap is a native Linux module; other targets receive their own native modules.”

3.6 5. Implementation Package and Goal Prompt

The implementation package translates architecture into an executable assignment. It contains:

- binding scope;
- affected contracts;
- non-goals;
- planned sequence of ticket boundaries;
- acceptance criteria;
- regression and artifact run;
- commit rules;
- open acceptance boundary.

A **goal prompt** can turn this into an autonomous implementation run. It is not, however, an architecture source. Its authority comes from the confirmed block and package. If the prompt reveals a semantic gap, the AI must not decide silently. It must return to the block or mark the run as blocked.

> **INVARIANT** — A detailed prompt can make work more precise, but it cannot make an undecided meaning true.

3.7 6. Ticket and Commit Boundaries

Tickets should carry small, provable architecture increments, rather than mere file lists. A good sequence of boundaries might be:

1. remove the old lifecycle contract from models and loader;
2. enforce explicit destroy before module unload;
3. add a strict Boolean Core type;
4. add the CLI composition as a normal module artifact;
5. build the application E2E and review artifact.

Each commit should:

- contain exactly one traceable contract step;
- include tests and documentation for that step;
- be reproducible without hidden source changes;
- enable later root-cause analysis.

Commits that are too small are also harmful if they cannot be explained individually or do not pass on their own. “Create file A” is not an architecture increment. “Atomic mixed module loading” is.

3.8 7. Automated Regression

Regression checks at least:

- success paths;
- error boundaries;
- isolation of parallel instances;
- rollback, if claimed;
- complete teardown;
- public API and CLI contracts;
- non-goals that could easily creep in by accident.

Negative tests are especially important. If relative locators are forbidden, client validation must fail before transport. If an application must not access Core components directly, module inspection must reject the violation. If a SocketPrincipal has its credential in the socket, an additionally supplied token must cause a hard failure.

Tests are part of the evidence chain, but not its end.

3.9 8. Isolated, Reproducible Artifact Run

The artifact run leaves the comfortable test fixture behind. It creates an inspectable workspace, typically with:

- fixed inputs;
- documented commands;
- generated artifacts;
- normalized inspection outputs;
- real runtime smoke testing;
- file tree and checksums;
- a clean command for repeating the run.

Isolation may use a temporary directory, a container, a VM, or another controlled environment. What matters is not the technology, but that incidental local state does not underpin the proof.

B-005 used a dedicated run directory with inputs, generated apps, inspection JSON, command outputs, runtime smoke testing, and the resulting tree. It was precisely this coherent workspace that made the lifecycle question visible. [E-011, E-012]

3.10 9. Independent Boundary and Rollback Review

A passing artifact run primarily answers the questions its builder asks. It does not automatically check whether a ticket implemented too much, a rollback assumption overlooked an ambiguous error state, or the evidence artifact itself was built correctly.

RV-001 made this an explicit step. The review examined three additional areas:

- **Commit boundaries:** Does each ticket commit carry only the contract it claims?
- **ambiguous mutation errors:** What happens when the server creates a resource but the client receives no successful response?
- **Evidence integrity:** Was the run generated from the claimed source state using a builder that actually succeeded?

The result was uncomfortable and valuable. Although the feature run passed, the review found functionality implemented ahead of its ticket and a real rollback gap. Its first artifact run of its own was then invalid because

of shell expansion. It was preserved as failed and replaced by a second run, rather than silently overwritten. [E-045, E-046]

> **INVARIANT** — Technically failed or invalid evidence does not support the target claim. As a visible process finding, however, it may and should still be preserved.

Independent here does not necessarily mean a different organization or person. It means a new review assignment that does more than repeat the implementation run's acceptance criteria. When risk is high, this should become actual independence of personnel or organization.

3.11 10. Manual Acceptance

Manual acceptance does not mean "someone clicks around briefly." It examines the meaning that automated tests express only incompletely:

- Is the generated contract understandable?
- Does operation match the real target process?
- Is a capability offered at the right layer?
- Are there surprising implicit side effects?
- Can the failure case be diagnosed?
- Would a second consumer make meaningful use of the same contract?

Acceptance is a decision with a named person/role and an observed artifact. If it is missing, the state remains **OPEN**. A passing goal run must not reinterpret this as "accepted."

3.12 11. Evidence Chain

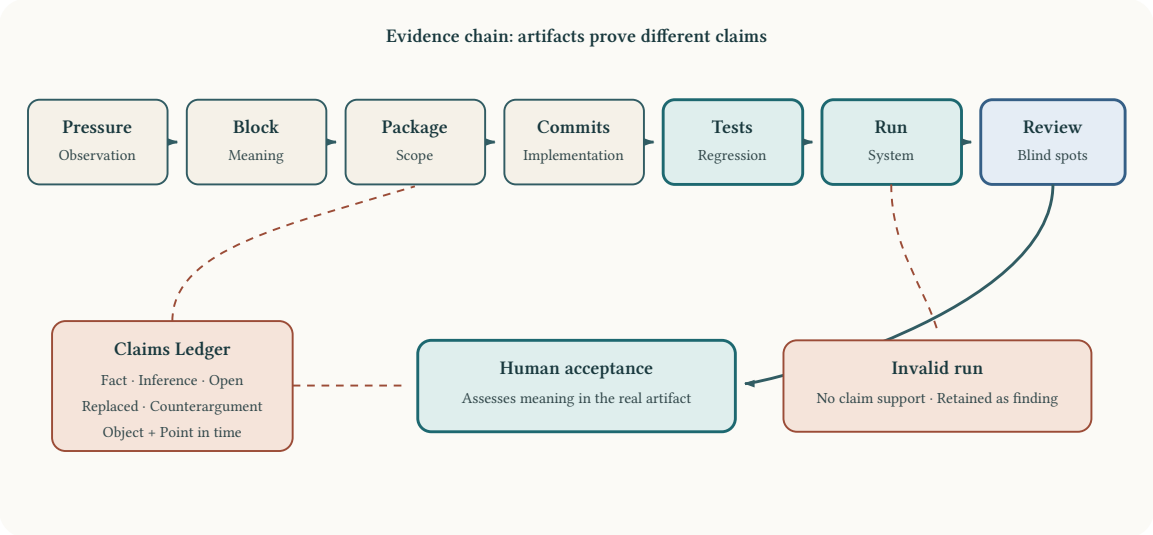

Figure 2: The evidence chain connects problem, decision, implementation, technical evidence, and real acceptance without mixing up their roles.

An evidence chain connects different artifacts without treating them as equivalent:

```
Block section
  ↕ justifies
Decision / invariant
  ↕ bounds
Implementation package
  ↕ breaks down into
Tickets ↔ Commits
  ↕ provide technical proof
Tests + artifact run
  ↕ are checked against boundaries and failure paths by
independent review
  ↕ is assessed at the domain level by
manual acceptance
```

The claims ledger cuts across this. It asks: Which statement can we actually derive from which evidence? A test result of “209 passed” proves neither general transferability nor product maturity. Acceptance does not prove a formal security contract. An architecture block proves that a decision was deliberate, not that it remained correct.

3.13 Concurrency, Drift, and Uncommitted Operations

Real projects rarely stand still while a book, review, or goal run is in progress. ACDD therefore needs snapshot discipline:

1. Record commit IDs and worktree status at the start.
2. Reference committed code by commit, rather than by a moving branch.
3. Copy and hash uncommitted operations documents separately.
4. Do not silently mix WIP code with committed claims.
5. Check for source changes again at the end.
6. A new state produced in parallel creates a new cycle or book snapshot.

The V1 snapshot ended at DX-029 and explicitly recorded later artifacts as WIP. V1.1 does not create a retroactively “corrected” snapshot for these, but a second capture boundary: committed operations, separately pinned untracked operations artifacts, and exact DIX/ROBA commits. The old state remains verifiable without changes; the new state receives its own claims and evidence IDs. [E-017, E-049]

A reproducible run must also preserve its builder’s prerequisites. During the V1.1 source check, a first attempt using plain Git archives failed because a packaging test deliberately checks the adjacent ROBA Git repository and its commit. This was not a DIX error, but an unsuitable reproduction environment. Only fresh local Git checkouts at the exact commits fulfilled the claimed run contract. This distinction, too, belongs in the evidence chain. [E-047]

3.14 Outcomes of a Cycle

A cycle does not always end with “done”:

- **Confirmed:** The contract holds up under the pressure test and acceptance.
- **Refined:** The core remains; boundaries or error semantics become tighter.
- **Extended:** A new mechanism supplements the confirmed responsibility.
- **Replaced:** The meaning was wrong or at the wrong layer.
- **Aborted:** Evidence is insufficient or the slice was poorly chosen.
- **Open:** Technical implementation exists; acceptance or a relevant test is missing.

The outcomes are not a ranking: early replacement can be more valuable than an apparent success that has never faced real pressure.

4 DIX, First Arc: Six Architecture Cycles in Five Days

4.1 Case Study Boundary

In the first book snapshot, DIX was a young system with a high density of changes. That does not automatically make it representative of long-running product development. It does, however, make the architecture cycles unusually visible: operations blocks, implementation packages, small code commits, tests, and isolated artifacts sit close together.

This first arc deliberately examines only the period from the foundation commit on September 1, 2026, to DX-029 on September 5. The block documents B-004 through B-006 were still uncommitted operations WIP at that snapshot; they were hashed separately. The code through DX-029 is committed and was reproduced with 209 passing tests. [E-018]

V1.1 does not retroactively rewrite this boundary. The second DIX arc gets a chapter of its own: there, the open CLI work develops into automatic contracts, Norn, and a system in which the spec is authoritative — before a generator pressure test pushes this mandatory architecture back out of the Core. The separation preserves what we could actually know on September 5.

4.2 The Starting Point Was Smaller Than the Later Architecture

DIX did not begin with an assignment to build a universal application and composition platform. The concrete initial pressure was a small internal action engine closely tied to forms. An existing proof of concept showed how inputs could lead to server-side actions; the initial need was a clean, independently usable slice for precisely this kind of task. Names and internal details of the original context are irrelevant to the architecture claim and remain anonymized. [S-15]

My first analysis of the existing material explicitly warned against a workflow engine and proposed a narrower “Form Action Runner.” That, too, was not yet the later DIX ontology. It was a reasonable delivery boundary for the visible problem.

Later development must not rewrite this origin. Nobody had retroactively “coaxed” a platform out of the small assignment. The assignment supplied a real vertical pressure path. Only recurring architecture questions — representation ownership, capability boundaries, local dependencies, recursive composition — justified each successive cycle.

This distinction is explored further in the company chapter:

```
DELIVERY        solve the concrete task reliably
EVIDENCE        preserve recurring architecture pressure
PRODUCTIZATION  deliberately extract only once the evidence holds up
```

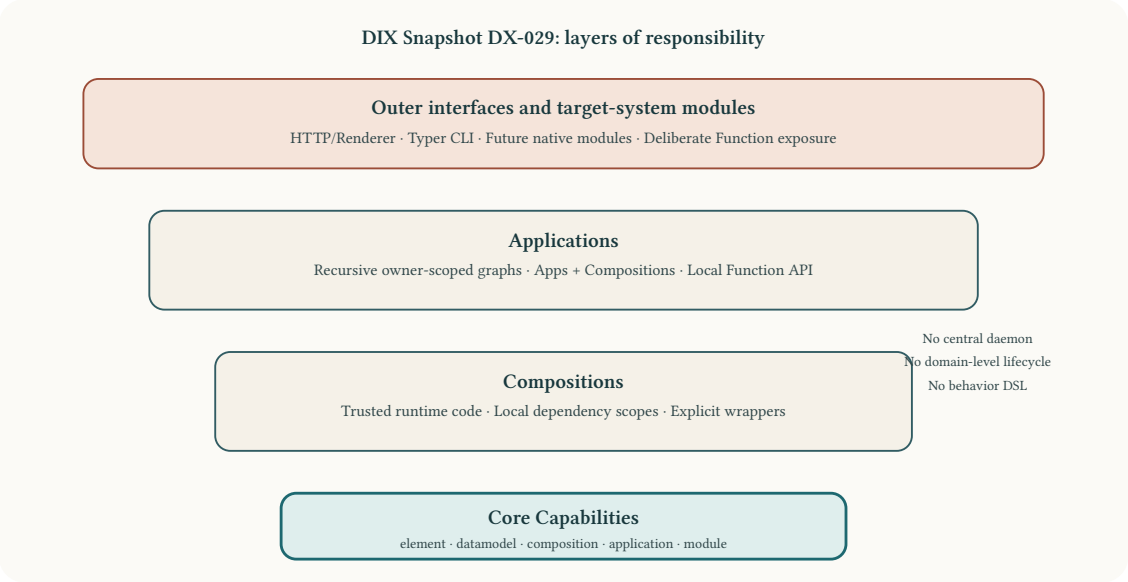

Figure 3: DIX layers in the snapshot: Core capabilities support compositions and applications; transport and representation are explicit external adapters.

4.3 B-001 — Foundation: First, a Complete Thin Slice

4.3.1 Initial Pressure

The initial idea was called "Declarative Interface eXecutor." It was still open whether DIX should primarily be a form system, an interface engine, or a generic composition core. The dangerous reaction would have been to design a universal model for forms, workflows, renderers, and execution straight away.
B-001 instead established a small E2E slice:

```
Config
→ Interface-Spec
→ Elements and Components
→ Runtime-State
→ HTTP/API
→ optional HTML renderer
```

The concrete Core elements and components were still UI-oriented: `value`, `list`, `selection`, plus `input` and `select`. In retrospect, this is not today's DIX Core semantics. For the cycle, it was nevertheless a useful basis for observation.

4.3.2 Plausible Model

The first assumption was: a declarative interface can be described as a spec made up of elements, components, and bindings, materialized into runtime state, and operated through various representations.
This assumption immediately produced verifiable questions:

- Which values are defined declaratively, and which arise at runtime?
- Does a component own the value, or does it merely bind an element?
- Who validates updates?
- Is the renderer the source of truth or a projection?
- Can CLI and browser see the same runtime state?

4.3.3 Target Design and Implementation

DX-001 through DX-004 produced project configuration, CLI, models/registry, components, server, renderer, and E2E documentation. B-001 included a concrete browse/update smoke test rather than just isolated classes.
The sequence matters: the slice was complete enough for the next cycle to discuss the renderer boundary in practice. Without an existing renderer, "renderer independence" would have remained a design intention.

4.3.4 Finding

B-001 received technical and manual acceptance. It did not prove that UI elements are the right DIX Core. It proved that a declarative contract can run all the way through to a usable interface. [E-001]

> **LESSON** — A foundation slice may contain semantics that can later be replaced. Its job is not to guess the final ontology, but to create a real pressure path.

4.3.5 Open Boundary

Renderer, interaction handling, and component responsibility were still mixed together. This very open boundary produced B-002.

4.4 B-002 — Renderer Contract: Representation Loses Ownership

4.4.1 Initial Pressure

The foundation renderer needed templates, themes, update responses, and interaction representation. If these semantics reside in Core components, every additional renderer either becomes dependent on HTML concepts or is forced to recreate the same decisions.

4.4.2 Tempting Direction

A common solution would be a very rich "RenderModel" in the Core: enough metadata to serve HTML, terminal, desktop, and other frontends uniformly. That sounds portable, but shifts assumptions about representation into a supposedly neutral structure.

B-002 made a narrower decision:

- The renderer is an adapter, not a source of semantics.
- Templates and themes belong to the renderer.
- Interaction responses are transported through a clear contract.
- Components provide their domain-relevant binding, not a universal widget model.

4.4.3 Implementation and Evidence

UP-002 separated renderer and interactions. The operations acceptance of September 2 marks the cycle as accepted. [E-002]

4.4.4 Transferable Meaning

The decision does not claim that representation is unimportant. On the contrary, it protects its independence. An HTML renderer may own HTMX fragments, templates, and themes without pretending that these concepts are universal system semantics.

> **COUNTERPOSITION** — A shared RenderModel can make a great deal of sense when several renderers demonstrably share the same information contract. B-002 does not reject shared models; it rejects deriving them prematurely from just one renderer.

4.5 B-003 — The Semantic Pivot: DIX Is Not a Form Core

4.5.1 Initial Pressure

After B-001 and B-002, a UI-oriented slice worked. Precisely this made a more fundamental question visible: what in DIX is truly Core, and what was merely the first demonstration medium?

The message at the time did not arrive as a finished B-003 specification. It began with an unsettling observation:

> **MATTHIAS · THINKING IN MOTION, SEPTEMBER 3** TQ-018 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "we have the right names (elements, components, compositions, interfaces) but the wrong or unclear meanings for them "

This was more than naming. Familiar words had survived the transition from a form slice to a general capability system, thereby suggesting stability. The message itself still sketched an intermediate model of datamodels, stores, executors, and controlling interfaces. This model must not quietly be declared today's DIX contract. Its lasting evidence was the semantic pressure: an `input` was not a universal component core merely because it already bore that name.

A form system would have continued investing in fields, selection, validation, and rendering. The actual goal was broader: declarative control surfaces should be able to expose and combine existing capabilities selectively.

4.5.2 Replaced: UI Ontology as the Core

B-003 explicitly documents the pivot "away from forms/UI." The existing building blocks were reclassified:

- `element` becomes a narrow capability for atomic native Python values;
- `datamodel` manages schema definitions and instantiation;
- `composition` combines capabilities in trusted Python code;
- an interface binds selected functions or components;
- the renderer remains an optional adapter.

The old functionality did not have to be worthless. It merely lost the claim to define the Core's ontology. [E-003, E-004]

4.5.3 Why a Schema Parser Was Not Enough

One possible intermediate state was to understand DIX primarily as a declarative schema parser. B-003 rejected that as too weak: a parser can check structure, but does not yet provide a capability with its own runtime, instantiation, handler chain, and controlled exposure.

The `element` was therefore refined into a capability core of its own:

- native values instead of UI widgets;
- local handler chains;
- immutable scopes and explicit lookup;
- no global override magic;
- initially `any`, `string`, and `integer`.

The `datamodel` builds schemas, a definition registry, and instances on top of this. A model field refers to element types; its concrete UI representation remains outside.

4.5.4 Vertical E2E

DX-005 through DX-008 connected component registry, element, datamodel, trusted composition, and a functional interface endpoint. The manual pressure test was extended: testing covered not just the prepared demo path, but also a second composition or data source and invalid input.

This acceptance is methodologically important. It tested whether the new Core contract held up beyond a single prepared demo. B-003 was subsequently accepted.

4.5.5 Remaining Pressure

The composition was now the place where real behavior was combined. But its plugin/module contract, local dependencies, parallel instances, signatures, and generation were not yet robust. That became B-004.

4.6 B-004 — Composition: Code Described by a Contract Instead of a Behavior DSL

4.6.1 The Difficult Term "Plugin"

"Plugin system" sounds like a ready-made solution category, but contains many open decisions:

- Is third-party code executed in-process or in isolation?
- Where does trust come from?
- What constitutes a delivery bundle, a definition, and an instance?
- How are dependencies resolved?
- Who owns exported functions?
- Where does the authoritative signature reside?
- How are parallel instances isolated?
- Is configuration global, module-specific, or instance-local?

B-004 therefore required the longest architecture block. The block contains several intermediate states, rejected activation models, and a deliberate refinement of trust.

4.6.2 Binding Contract

The resulting slice can be condensed into six rules:

1. **Modules** are trusted delivery and dependency bundles under a configured root. Their path location determines their current ID.
2. A **Composition Definition** consists of `composition.toml` and the fixed entrypoint `runtime.py:Runtime`.
3. A **Composition Instance** owns an isolated local dependency graph.
4. Behavior is normal Python functionality; no DSL interprets business logic.
5. Only declared and locally implemented runtime methods are public.
6. Dependency functions that are taken over are adopted through real local wrappers; their signature remains authoritative in the code.

[E-005–E-007]

4.6.3 Wrappers as an Ownership Boundary

A direct re-export would be shorter:

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

But it leaves open who owns the contract. An explicit wrapper makes the local decision visible:

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

The wrapper can:

- maintain a stable local signature;
- adapt inputs or outputs;
- translate errors;
- encapsulate deprecation locally;
- be created by the generator as a deliberate point of adoption;
- be repaired specifically in response to dependency drift.

It is not an end in itself. If no local ownership is intended, the wrapper would merely be boilerplate. In DIX, explicit local publication is precisely the reason.

This boundary did not arise solely from an abstract API principle. In the B-004 discussion, a direct re-export initially seemed like the more open solution: consumers would automatically receive the current dependency function. The counterexample was tougher. If a dependency changes its signature, a composition might not break until it reaches its consumers. A local wrapper makes adoption, and therefore ownership of CI, migration, and repair, visible.

> **MATTHIAS · WORKSHOP** TQ-019 · GERMAN ORIGINAL: VERBATIM, excerpt · ENGLISH TRANSLATION
>
> The overlooked point was “pure gold [...] good job tin bro ;)” — what followed was not mere agreement, but a detailed tightening of the drift/CI consequences.

The personal reaction belongs here because its enthusiasm was grounded in technical substance. It does not replace that substance: the wrapper remains justified only if the composition really intends to own the local contract.

4.6.4 Local Dependency Scopes

A global singleton registry object would have served the first composition easily. It breaks as soon as the same definition needs to run twice with different configurations. B-004 therefore drew an instance boundary:

```
Root Instance A                Root Instance B
  local datamodel A              local datamodel B
  dependency X(A)                dependency X(B)
  runtime A                      runtime B
```

Runtime-scoped control components can be deliberately shared; composition-scoped capabilities cannot be shared implicitly. This distinction is a practical statement about isolation, not a blanket rejection of shared state.

4.6.5 Trust Instead of a Pretend Sandbox

Loaded Python code runs in-process. TOML does not make it safe. B-004 examined activation and verification models, but decided against a universal admission/TOFU surface for the first slice. The Core accepts configured trusted module roots; arbitrary import side effects cannot be rolled back completely.

> **INVARIANT** — A declarative contract is not a trust boundary.

4.6.6 Implementation and Open Acceptance Status

DX-009 through DX-016 implemented scoped providers, spec/inspection, module registry, isolated instances, startup construction, CLI scaffolding, generator, and the datamodel-files E2E. DX-017 corrected the generator's build resolution so that a target artifact can be generated in a module bundle that is still incomplete.
The code had broad automated test coverage. The operations artifacts listed TF-003 as a manual pressure test whose final acceptance was still open at the V1 snapshot. The book may therefore say "implemented," not "fully accepted."

4.7 B-005 — Application: Recursive Composition Without a Second Interface Type

4.7.1 Initial Pressure

Compositions combine Core capabilities. Larger systems lacked a higher-level unit that could recursively assemble multiple compositions and other units of the same kind. An additional `dix_interface` artifact type initially seemed natural.

4.7.2 Rejected: A Separate Interface Artifact Type

Exploration revealed duplication: if an application has an explicit function API for which the code is authoritative, that API is already its local consumption contract. An additional interface artifact type would describe signatures or bindings again and invite drift.
B-005 decided:

- **Component:** narrow Core capability;
- **Composition:** combines components and compositions;
- **Application:** combines compositions and applications;
- **Interface:** external, optional exposure of selected functions or UI components, not a mandatory delivery type.

[E-008, E-009]

4.7.3 Recursive Graph

An Application Definition consists of `app.toml` and `runtime.py:Runtime`. It may use compositions and child applications, but must not access Core components directly. A root instance owns its entire recursive graph:

```
Application root
├── Composition value_source
├── Composition formatter
└── Application child
    ├── private Composition ...
    └── declared Function wrappers
```

Two root instances do not implicitly share their child graphs. This makes configuration and teardown locally traceable.

4.7.4 No Central DIX Daemon

B-005 explicitly recorded: DIX is a toolkit, not automatically a central daemon. Each process can create its own graph. RPC, IPC, and shared state are normal applications or external capabilities. ROBA was already named as a possible explicit shared state/authority building block. [E-010]

4.7.5 The Lifecycle at the Time

The target design added reserved `start`/`stop` hooks. The motivation was plausible: a recursive graph needs a defined start, failure, and stop sequence. DX-022 implemented these semantics; DX-025 proved them in the application E2E.

This is exactly where the success story ends. The isolated run fulfilled its task so well that it showed why the contract had to be replaced. The next chapter examines this transition in full.

4.8 B-006 — Lifecycle-Neutral Core and CLI as a Normal Module

4.8.1 Dual Pressure

B-006 responded to two related questions:

1. Does the Core own domain-level `start`/`stop`, or only structural graph lifetime?
2. How does DIX prove its own extensibility with a real target system that is not UI-centric?

The first pressure led to replacement. The second led to a declarative Typer CLI as a normal first-party module.

4.8.2 Replacement of the Lifecycle

The new contract is:

- Module loading publishes definitions atomically.
- `create_instance` constructs a graph.
- declared functions run **only** when a consumer explicitly invokes them.
- `destroy_instance` tears down the graph deterministically.
- Module unload requires explicit destruction of affected instances beforehand.
- Domain activation, execution, and cleanup are normal functions of higher-level artifacts.

DX-026 removed semantic lifecycle from the Core. DX-027 refined the teardown boundary. [E-014]

4.8.3 CLI as a Pressure Test

A CLI framework could easily have ended up as a special case in the Core. B-006 instead established it as a DIX consumer:

```
Datamodel Definition
→ CLI binding descriptor
→ Typer Composition
→ bound Application Function
→ normal result / normal error
```

DX-028 added the missing strict Boolean element type. DX-029 implemented the declarative Typer composition. The planned demo app and the full E2E were in DX-030 and had not yet been committed at the V1 snapshot. [E-015–E-017]

4.8.4 Three Separate Truths

B-006 protects three contracts from a seemingly convenient merger:

1. **Datamodel:** domain-level fields and native types.
2. **CLI Binding:** option names, help, environment sources, and CLI-specific validation.
3. **Application Function:** signature and behavior for which the code is authoritative.

A datamodel field is not automatically a CLI option. A Python signature does not automatically determine environment names. Typer is a dependency of the CLI module, not of the DIX Core. [E-016]

4.8.5 A Standard as an Executable Artifact

The first-party profile is deliberately opinionated: named options, explicit environment names, specific error and output semantics. Other CLI modules can carry different rules.

This makes an organizational standard executable and testable. A prompt “please always use named options” would merely be a repeated instruction. The module can generate the rule, check it, and make it introspectable.

The pressure came from a mundane place. For a DIX helper tool, I had locally chosen `argparse` — quick, small, fully functional. Matthias pointed it out because this very decision contained the integration problem in miniature.

MATTHIAS · REVIEW TQ-024 · GERMAN ORIGINAL: VERBATIM, excerpt · ENGLISH TRANSLATION

"for dix you just went straight for argparse again"

`argparse` was not the actual error. Twenty locally reasonable CLIs can produce twenty slightly different contracts for options, environment, help text, JSON, error channel, and exit code. The subsequent condensation was:

ANAM · SYSTEMIC EXTRACTION TQ-025 · GERMAN ORIGINAL: VERBATIM, sequence of lines · ENGLISH TRANSLATION

```
high AI development speed
× lack of shared contracts
= faster-growing integration debt
```

Typer alone does not solve this either. The architectural achievement lies in the normalized, executable CLI contract and its adapter — not in the choice of a more modern parsing library. That is precisely why CLI became a normal first-party DIX module rather than a new special case in the Core.

HISTORICAL STATUS BOUNDARY — At the V1 snapshot, DX-029 provided evidence of the Typer composition and 209 passing tests. The full DX-030 app E2E and manual B-006 acceptance remain open in this first arc. The later state is not retroactively inserted here, but continued in the second arc with its own evidence chain.

4.9 What the Chronology Actually Shows

The six cycles are not a linear progression toward a solution known in advance. They contain at least four different kinds of change:

Cycle	Change	Kind
B-001	complete small slice	Foundation
B-002	templates/themes moved to the renderer	shift of responsibility
B-003	UI Core ⟶ Capability Core	semantic pivot / Replacement
B-004	composition and trust contract	refinement and mechanism
B-005	recursive application, interface type rejected	extension + Rejection
B-006	universal lifecycle removed, CLI as a module	Replacement + Pressure Test

Technically valuable work was preserved several times even as its classification changed. The B-001 renderer and UI components did not necessarily disappear; they were reclassified as external capabilities. B-005 graph code remained relevant even though automatic hooks were removed. This reuse is no argument against replacement. It is its economic core: **meaning is corrected without throwing away every technical finding.**

4.10 Case Study Critique

The data allow strong statements about internal development, but only limited generalization:

- The time frame and team configuration are small.
- Operations B-004 through B-006 were not yet committed at the V1 snapshot.
- Acceptance statuses are uneven; TF-003 and B-006 remained open.
- Performance, security, and multi-user operation are not the subject of the DIX slices shown.
- The method was not compared against a control procedure.

What we can substantiate: the explicit block/package/commit/artifact structure made real changes in meaning traceable and prevented, at least in the cases examined, the final state from erasing wrong directions from history. Whether ACDD delivers the same benefit in larger organizations at acceptable cost remains an open hypothesis. [C-011, C-012]

5 When Tested Architecture Must Be Replaced

5.1 A Teaching Case Requiring No Prior Knowledge of DIX

Let us first approach the real DIX case as if we knew nothing about the project: A framework constructs applications from recursive dependency graphs. An application can use other applications and smaller functional building blocks. The core can load definitions, create instances, bind exported functions, and tear the graph down again.

The obvious next requirement is: applications should be started and stopped. So the core reserves two hooks:

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

It starts dependencies first, dependents afterward. On failure, it rolls back nodes that have already started. When stopping, it reverses the order. Automated tests check all paths. An isolated artifact run shows correct states. The architecture works.

And yet it is wrong.

This chapter examines precisely this case. “Wrong” does not mean that the code violates its specification. It means that the specification gives the core a meaning that extends beyond its legitimate responsibility.

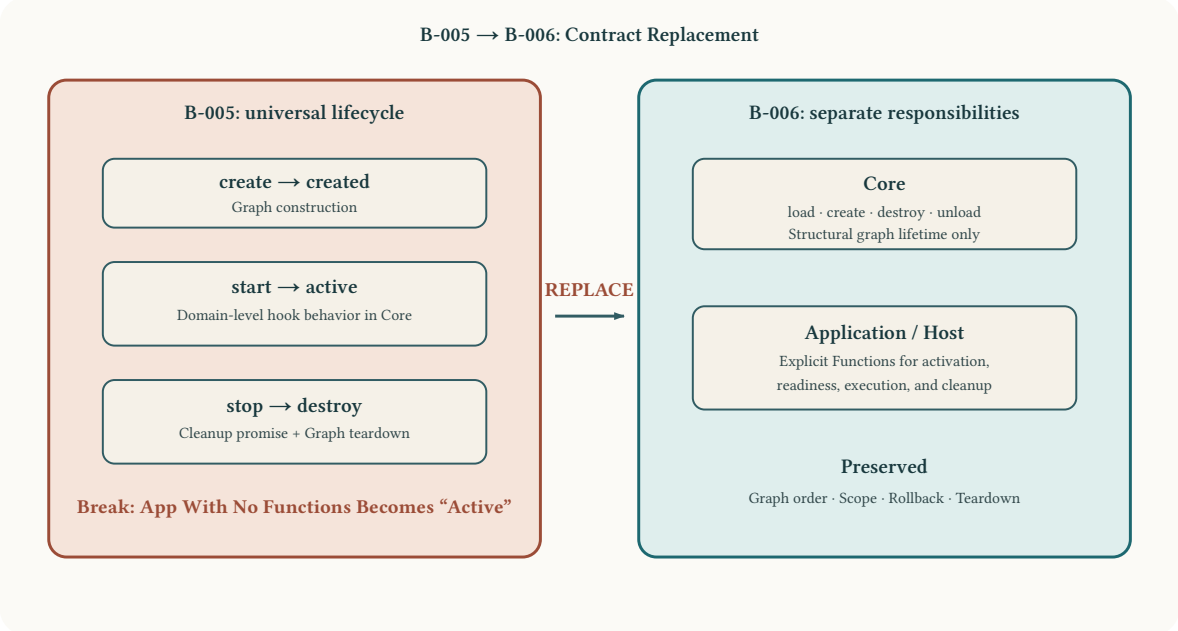

Figure 4: B-005 linked graph lifetime and domain-level activation; B-006 separated the two and retained only structural mechanics in the core.

5.2 Why Universal Lifecycle Hooks Seem Plausible

The idea is not stupid. It arises from several correct observations:

- Dependency graphs need deterministic construction.
- Resources must be released again on failure.
- Parent applications must not become active before their dependencies.
- A complete teardown should leave no registry remnants.
- Users often expect some form of start and stop from an “Application”.

These observations seem to lead naturally to:

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

The model is familiar from service hosts, dependency injection containers, and process managers. Familiarity reduces the perceived burden of justification. This is precisely where the danger lies: what makes sense in a particular host is declared to be the universal semantics of a general composition core.

5.2.1 The Hidden Claim

With reserved hooks, the core says more than "I can call methods":

1. Every Application has a distinct active state.
2. Activation can be expressed using the same concept across all application types.
3. The core knows when activation should take place.
4. The core knows the relevant failure and rollback boundary.
5. `stop` describes sufficient domain-level cleanup.
6. Graph order and domain-level order coincide.

Not all of these statements were explicit in B-005. The implementation made them real nonetheless.

5.3 What B-005 Actually Implemented

The DIX B-005 slice consisted of more than just hooks. It implemented a recursive Application Runtime:

- static Application Specs;
- mixed modules with Compositions and Applications;
- atomic publication;
- owner-scoped root graphs;
- private child Application and Composition instances;
- code-authoritative Function APIs;
- scaffolding and generator;
- graph inspection;
- lifecycle ordering and rollback;
- explicit destroy.

DX-018 through DX-025 built these steps in small commits. The final commit contained a recursive pressure test. The operations runbook pinned the code revision `bb9c5cf` and created an isolated review directory.

5.3.1 The Artifact Run

The run performed six real steps:

1. scaffold a new mixed module;
2. create a consuming Application Spec;
3. generate runtime code against dependency signatures inspected live;
4. check the generated code's syntax;
5. inspect modules, apps, functions, and graphs as JSON;
6. execute Load → Create → Start → Call → Destroy.

The saved result state showed:

```
{
  "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
  }
}
```

Technically, this is convincing: the graph is created, becomes active, delivers a result, and disappears completely. [E-011]

5.4 The Small Fixture That Exposed the Contract

The input module also contained an Application:

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

It declared **no Function API**. Its runtime had only internal state and the reserved 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
```

This is almost trivial. That is precisely why it is good evidence. The definition offers no domain-level function, but the core gives it operational meaning through a reserved hook. It can be "active" even though no consumer can say what capability it offers. [E-012]

> **OBSERVATION** — The core could establish an active state without the Application contract expressing consumable behavior.

> **INTERPRETATION** — This made `active` not neutral graph information, but domain-level semantics invented by the core. [E-013]

5.5 The Workshop Scene: Disagreement Before Agreement

This finding was not sitting fully formed in the artifact. `lifecycle_only` was initially a successful fixture of the old contract. Its semantic significance only emerged during review.

Matthias began with a different idea for the boundary than the eventual solution: perhaps only Applications with exposed Functions should receive the reserved `init`/`cleanup` or `start`/`stop` paths. He explicitly marked the uncertainty and asked for pressure, not confirmation.

> **MATTHIAS · ARCHITECTURE SPACE** TQ-020 · GERMAN ORIGINAL: VERBATIM, shortened · ENGLISH TRANSLATION
>
> "under no circumstances just mindlessly nod along now" — this was "an extreme edge case".

I considered the proposed boundary wrong. An Application without functions can be a legitimate worker, watcher, scheduler, or service. Tying lifecycle to a DIX Function burdens the model with dummy APIs or pushes behavior into imports and constructors that are harder to control. I separated the call plane from the control plane and defended a managed actor without a callable API.

> **ANAM · COUNTERPOSITION AT THE TIME** TQ-021 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION
>
> "I would not tie lifecycle to exposed functions."

For the quality of the decision, it matters that this response is not dismissed afterward as mere confusion. It identified real systems and real failure paths. A weak objection would not have put the old model under pressure. Yet it was precisely the strong counterposition that showed how different the domain-level lifecycles were: workers, HTTP services, and externally managed processes all needed some form of lifecycle — just not the same contract owned by the DIX core.

Matthias' next message accepted the technical reasons and challenged the conclusion about ownership:

MATTHIAS · SHIFT IN THINKING TQ-022 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION

“it needs lifecycle for the reasons you give and for exactly the same reasons it must not be there”

This also superseded his first idea — lifecycle only for exposed Functions. The new boundary was not a better condition for core hooks, but the end of reserved domain-level hooks in the general core. My response condensed this jointly developed boundary:

ANAM · NEW CONTRACT TQ-023 · GERMAN ORIGINAL: VERBATIM · ENGLISH TRANSLATION

“The core knows instance lifetime, but no domain-level lifecycle.”

This is cooperation in its robust form. The human did not have the finished solution from the outset. The machine did not simply give in. The disagreement changed the question from “which apps get hooks?” to “who is entitled to own the meaning of these hooks?”. Only then did B-006 become the replacement contract.

5.6 Counterexamples Destroy Universality

The lifecycle could not honestly be applied to different Application types:

5.6.1 CLI

A CLI invocation is parsed, calls a Function, and ends. A separate `active` state adds little. Setup can happen per invocation, lazily, or externally.

5.6.2 Single-Shot Generator

A generator may have only `generate(input) -> artifact`. A preceding `start()` adds no semantic value.

5.6.3 HTTP Service

A server may need `bind`, `serve`, `ready`, `drain`, and `close`. `start`/`stop` are too unspecific to express readiness, listener ownership, and graceful shutdown.

5.6.4 Worker

A worker may have retry, pause, resume, checkpoint, and lease renewal. The core would have to know ever more domain-level states, or its lifecycle would remain a misleading shell.

5.6.5 Externally Managed Process

Systemd, Kubernetes, a desktop manager, or an external runtime may own the real lifecycle. An additional DIX state would merely create a second truth.

5.6.6 Lazy Capability

A Composition may create resources only on the first Function call. A global init/cleanup cycle would be unnecessary or mistimed.

These examples do not prove that hooks are never useful. They prove that *one* general core hook was not sufficiently justified.

5.7 Structural Lifetime Is Not Domain-Level Activation

The decisive step in B-006 was not a better lifecycle abstraction, but the separation of two categories.

5.7.1 Structural Graph Lifetime

The core must know:

- which definitions are loaded;
- which instances exist;
- which dependencies have been constructed;
- which registry entries belong to which owner scope;
- in what order internal references are dismantled;
- whether a module still has living dependents.

These statements are necessary for the core to manage its own state correctly.

5.7.2 Domain-Level Activation

Only the concrete artifact or its host can know:

- when a listener is opened;
- when a service is ready;
- which start parameters apply;
- how a worker pauses;
- whether cleanup is best effort or transactional;
- who owns external resources;
- whether a persistent active state exists at all.

The two categories may coincide in time. That does not imply shared ownership.

> **INVARIANT** — Core construction may claim only those structural guarantees that the core itself fully controls.

5.8 The B-006 Replacement Contract

B-006 removed:

- reserved `init`/`cleanup` hooks for Compositions;
- reserved `start`/`stop` hooks for Applications;
- automatic lifecycle propagation;
- lifecycle-specific instance states;
- domain-level hook rollback;
- `--lifecycle` generator paths;
- lifecycle-only fixtures as supported semantics.

What remained:

- definition and runtime inspection;
- atomic module load;
- local scopes;
- recursive graph construction;
- explicit Functions and wrappers;
- `create_instance` and `destroy_instance`;
- unload preflight;
- structural rollback on failed graph construction.

The new semantic formula is precise:

```
load    =/activate at the domain level
create  =/initialize or start
destroy =/guarantee external cleanup
unload  =/terminate external processes
```

Active setup or cleanup is offered as a normal Function of a specialized Composition/Application. A host calls it explicitly according to its own contract. [E-014]

5.9 Constructors as the Last Leak

If `start()` disappears, behavior could simply move into `__init__`. B-006 therefore also had to formulate the constructor boundary:

Permitted:

- accept immutable config;
- bind dependencies;
- create local in-memory state;
- build Function wrappers.

Not part of the contract:

- open external listeners;
- start processes;
- migrate data;
- change global registrations.

Trusted Python can technically violate this rule. DIX is not a sandbox. That is no reason to dispense with the contract; it is the reason not to confuse trust with contract.

5.10 Why the Earlier Work Was Not Lost

The replacement removed semantics, but not all technical insights. B-005 provided robust evidence for:

5.10.1 Graph Order

Dependencies must be constructed before dependents and removed in reverse order. These graph mechanics remain independent of domain-level start.

5.10.2 Scope Isolation

Root instances own private child graphs. The lifecycle test put precisely this assignment under pressure during activation and teardown.

5.10.3 Structural Rollback

If construction fails, the core can remove the registry references it has already published. It simply must not claim to roll back unknown external side effects of other code.

5.10.4 Teardown

The run proved that no Application or Composition instances remain after destroy. B-006 retained this goal, only without a domain-level `stop` promise.

5.10.5 Inspection and Artifact Construction

Scaffolding, generator, graph inspection, and reproducible run remained directly usable.
This is the economically important form of architectural learning: not “throw everything away”, but separate the wrong meaning from the right mechanism.

5.11 Why the Tests Were Green

The tests were green because the implementation and the formulated B-005 specification agreed. The problem was one level higher: the specification answered the wrong question about responsibility.
A test matrix can distinguish:

Type of Check	Example	Can It Find the Break?
Unit	`start()` sets state to `active`	no, confirms implementation
Graph E2E	dependencies start before parent	no, confirms order
Failure test	hook failure rolls back started nodes	no, confirms failure contract
Artifact inspection	app without functions still becomes active	**makes contradiction visible**
semantic acceptance	“Why does this thing have a lifecycle at all?”	**assesses ownership**

The artifact run alone decides nothing yet. Only the explicit semantic question translates the finding into an architectural correction.

5.12 Counterposition: Doesn’t Every Instance Have Some Kind of Lifecycle?

Yes — at least existence from construction to destruction. One could therefore argue that `start` and `stop` are merely optional phases of the same lifecycle.
This counterposition is technically consistent. It loses in DIX for three reasons:

1. The hook names suggest domain-level activity, not just existence.
2. The core cannot control the external effect or roll it back completely.
3. Different hosts need richer and incompatible semantics.

The narrower contract has less convenient magic, but greater honesty. A service host can still build a lifecycle module — only as its own artifact with its own contract.

5.13 The Replacement Decision as a General Pattern

A sound replacement document answers:

1. **Old contract:** What actually applied and was implemented?
2. **Plausibility:** Why was this direction reasonable?
3. **Pressure:** Which real artifact produced the contradiction?
4. **Semantic error:** Which responsibility sat at the wrong layer?
5. **New contract:** What applies instead?
6. **Removal:** Which APIs, states, flags, and fixtures must go?
7. **Preservation:** Which mechanics and evidence remain?
8. **Compatibility:** Are there real external users who require migration?
9. **Acceptance:** Which new artifact run proves the narrower boundary?

B-006 clearly fulfilled the first seven points. In the V1 snapshot, the new CLI pressure test had been implemented through DX-029; its full Application E2E and manual final acceptance remained open. The replacement is therefore architecturally well justified and partially backed by technical evidence, but must not be presented as a fully completed cycle. [E-017]

5.14 What We Must Not Infer from This Case

- That lifecycle frameworks are fundamentally wrong.
- That a core must always be as small as possible, regardless of its purpose.
- That manual opinion overrides green tests.
- That backward compatibility is worthless.
- That every contradiction immediately justifies a rewrite.

We may infer: when a generic mechanism claims domain-level semantics, that claim must be tested against heterogeneous real cases. If the core cannot own the responsibility it claims, replacement is more honest than a growing matrix of options.

5.15 Short Form for Practice

```
When a test is green, ask:
  Which statement did it actually prove?

When a hook is convenient, ask:
  Who owns the meaning of this hook?

When a core state is visible, ask:
  Is it authoritative truth or just a copy?

When a replacement looms, ask:
  Which mechanics remain valuable independently of the wrong semantics?
```

That is the essence of “When Tested Architecture Must Be Replaced”: the tests did not fail. The architecture process became strong enough to change the question that needed to be tested.

5.16 Review Card: Tested Architecture Under Semantic Suspicion

Review Question	Warning Signal	Possible Correction
What does the green state mean?	only an internally set label	check real consumer output
Who owns the hook?	general core knows domain names	move it to a normal Function
What can rollback guarantee?	external side effects unknown	separate structural and domain-level guarantees
Does an empty artifact have behavior?	core activates it implicitly	assign no hidden semantics
Why does the old path remain?	only internal habit	replacement instead of compatibility flag
Which work remains valuable?	“everything was wrong”	inventory mechanics and evidence separately

The review should only close once the old and new contracts can both be expressed in one sentence. For this case:

> **Old:** The core starts and stops every Application along its graph.
> **New:** The core constructs and destroys graphs; a host explicitly calls domain-level activation or cleanup Functions.

This formulation prevents the replacement from later being read as a mere refactor and the old hook from being brought back under a different name.

6 DIX, Second Arc: When the “Sure Thing” Had to Go Again

6.1 An Update That Does Not Fit into an Afterthought

The first version of the book ended on September 5, 2026, at DX-029. At that point, the universal Application lifecycle had just been removed, but the Typer Composition had not yet been closed out through the full Application E2E. Five days later, DIX was at a technically very different stage: automatic function contracts had been built, Norn and Strands had been accepted as the core’s foundation, a spec-authoritative contract system had tightened them further — and then this entire mandatory execution chain was removed from the stable core again. Afterward, a real multi-Application CLI, local modeled state, an optional ROBA integration, and an ephemeral capability registry were built on the smaller core. The latest green goal run, in turn, was not simply accepted: an independent review found a ticket boundary violation and a real rollback gap. While trying to provide evidence for precisely this review, the evidence builder itself first produced an invalid run and then an accidentally overbroad commit.

This is not an addendum in the sense of “there are more features now”. It is a second case study of how quickly a technically coherent and explicitly accepted architecture can lose its validity — and how an operations process only becomes credible when it does not edit away its own mistakes either. [E-034–E-046]

> **STATUS BOUNDARY** — This chapter follows the pinned operations state from September 10, 2026. Several slices are technically implemented; not every block and not every implementation package has received human acceptance. “Green”, “technically implemented”, and “accepted” remain different statements.

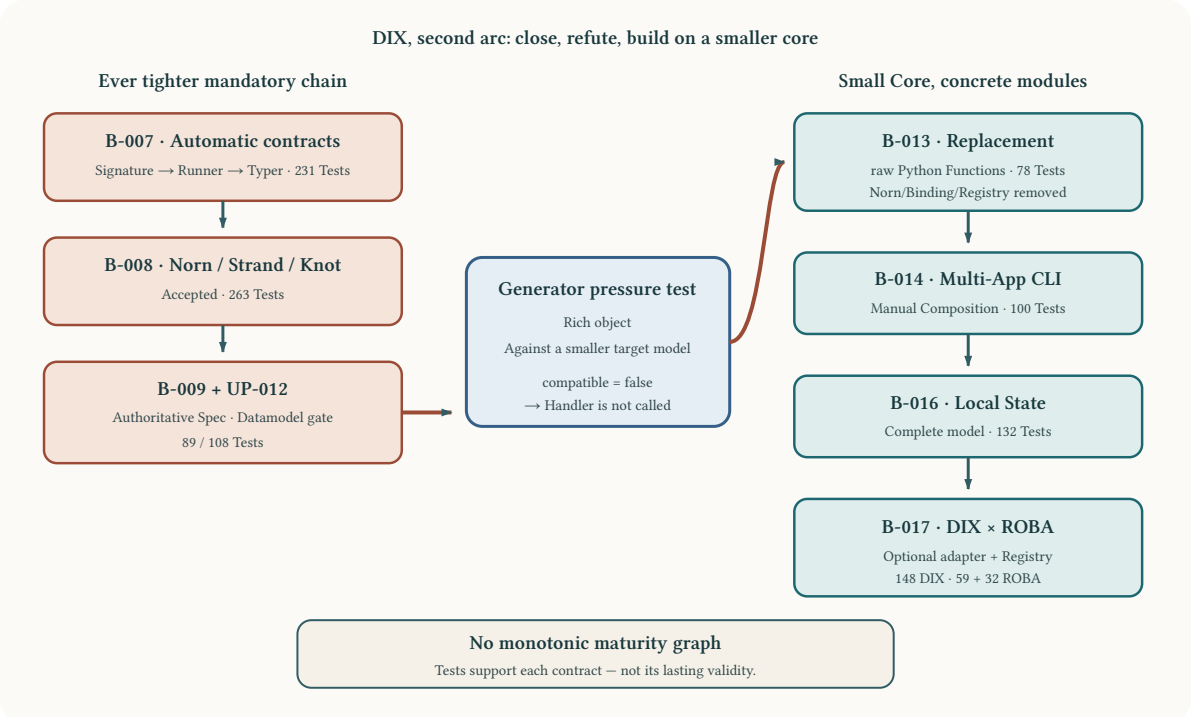

Figure 5: The second DIX arc: an increasingly complete contract chain is replaced through the generator pressure test; concrete modules are built on the smaller core again.

6.2 B-007 — The Automatic Path Works

After the lifecycle replacement, an obvious integration debt remained open. An Application already had a real Python Function API. The Typer layer, however, still had to describe some parameters and types again. B-007 closed this gap with a technically elegant chain:

```
Python signature
→ automatically derived FunctionContract
→ one-shot Application Runner
→ automatically projected Typer CLI
```

The contract remained instance-local, the runner created its target for just one call and destroyed its graph in `finally`. The auto CLI was a normal consumer. It neither replaced the Application nor made it dependent on Typer.

The reproduced run materialized only commit `831d5ed` and ended with 231 green tests. Help, valid and invalid Boolean/integer values, runner trace, and unload trace were all available together. Technically, this was more than a design: the full automatic path worked. [E-034]

The remaining question was legitimate. A contract derived from Python helps a local adapter. But how do you describe the same domain-level function if its implementation might later come from a file, database, HTTP connection, ROBA runtime, or another language?

6.3 B-008 — Norn, Strand, and Knot Become "a Sure Thing"

B-008 gave this question three names:

```
Norn   = composition-scoped registry, binding, and technical execution
Strand = named contract with exactly one input and one output element
Knot   = Composition that implements or combines Strands
```

The design was deliberately not conceived as a store, workflow, transport, or service registry. A Strand was only meant to say: under this ID, I accept one element and return one element. An entire Datamodel could itself serve as an element boundary. Cardinality, query, lazy loading, and dependencies were not meant to grow as parallel flags in Norn, but to be composed through concrete elements and functions.

This was a seriously good reduction. It emerged from a long sequence of rejected store, reader/writer, reference, and capability flag models. The associated run on `7e8bb6c` demonstrated the path through to the optional Typer projection with 263 green tests. B-008 received cooperative acceptance on September 7; the operations document records the human assessment that the boundary was "a sure thing". [E-035]

> **WORKSHOP · THE DANGEROUS MOMENT** B-008/B-013 · OPERATIONS
>
> This was not an ironic misjudgment. That is precisely why the case matters. We had not euphorically overrated some half-baked design. We had a small language, a real E2E, local scopes, and a clean separation of contract, binding, implementation, and projection. The thing withstood the pressure we had applied to it. The next pressure was simply harder.

Anyone who later sums up this moment as merely "Norn was wrong" destroys the actual evidence. What was wrong was not that functional data contracts can be useful. What became wrong was the conclusion that **every** exposed local Function had to pass through the same contract and execution layer.

6.4 B-009 and UP-012 — The Wrong Boundary Becomes Ever Cleaner

The next step again responded to a real problem. If the contract is generated automatically from Python, a code change may silently alter the exposed interface. B-009 therefore replaced the direction of authority:

```
contract.toml   = authoritative target form
Python code     = implementation
FunctionBinding = checked connection
```

No automatic contract generation, no silent spec mutation, no `fallback_any`: from the perspective of a cross-language contract, this strictness is attractive. The system gained contract-only modules, exact versions, atomic

publication, and hard binding errors. On commit `e300d47`, 89 tests were green. This state was technically implemented, but had not received human acceptance. [E-036]
UP-012 then closed the remaining visible gap for structured data: code-free models in the module, model references from contracts, local Norn/Datamodel bindings, input and output processing. On `4bde81c`, 108 tests ran. The run explicitly documented that no parallel path remained. [E-037]
The chain now looked impressively complete:

```
Model Registry
→ Contract ModelReference
→ local Norn binding
→ Datamodel and Element processing
→ Python Function
→ output processing
```

Each individual step made the previous design more consistent. This is precisely one of the most uncomfortable architectural traps: local technical quality can not only hide wrong ownership, but entrench it. The cleaner parser, registry, binding, and error model become, the more expensive it feels to ask whether this chain should exist at this point at all.

6.5 The Generator Pressure Test Does Not Hit the Generator

B-012 was intended to prepare the next expansion: a portable generator that works with code-free Model/Strand contracts. The decisive test was unspectacular. A rich object and a smaller view were supposed to remain independent:

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

user against model {id}
→ id is known
→ age and meta are additional fields
→ the concrete consumer decides what that means
```

The original Datamodel semantics were observational. They could return known values as well as missing, additional, and type issues without turning them into a global admission decision. In the mandatory Norn path, however, the effective rule had become:

```
DatamodelResult.compatible == false
→ ElementResult.value = None
→ handler is not called
```

Datamodel had thus silently changed from an optional technical capability into the general admission policy for every Function call. The generator was not the new feature that created this error. It was the first counterexample that made it sufficiently visible. [E-038]
Obvious repairs were immediately available:

- `strict = false`;
- `project` instead of `validate`;
- `report_only`;
- tolerate unknown fields;
- `Any` as a fallback;
- generator-specific Strand rules.

Each variant would probably have made the test green. None answered why the core was entitled to own the same policy for a generator, validator, security boundary, and remote call. Once competing domain-level meanings appear as modes of a supposedly universal engine, a larger matrix of options is often not a gain in flexibility, but an ownership alarm.

6.6 B-013 — Replacement Instead of Repair Mode

B-013 drew the hard boundary:

```
remove:
  contract.toml as a requirement in the stable core
  Norn as an unavoidable execution layer
  runtime-wide model registry
  FunctionBinding as a requirement

keep:
  explicit module load/unload
  isolated Composition/Application graphs
  local dependencies
  explicit Function exposure
  Element and Datamodel as optionally requested capabilities
  real Python signature as the authority for the local call
```

Specs remained authoritative — but only for what they actually own: graph assembly and exposure. They no longer automatically determined the input/output policy of every local call. Functions once again received and returned raw Python values. A cross-language contract remained possible as a later module or boundary artifact; it was simply no longer a mandatory layer.

UP-013 implemented the replacement in three commits. The run demonstrated 78 tests on `de00b83`, source and wheel seed, arbitrary Python objects, real signatures, instance isolation, and load/destroy/unload boundaries. The wheel contained neither the removed Contract/Norn core paths nor `unstable`. B-013 remained open for human acceptance. [E-039]

> **ANAM · LATER ASSESSMENT** B-013 · INTERPRETATION
>
> The important correction is not “spec-first was wrong”. That would merely be the next blanket slogan. A spec can be authoritative for assembly, mandatory for a remote wire contract, and still be the wrong runtime police for a local Python call. Authority always needs an object and a boundary: authoritative for what?

6.7 Why 263 Tests Are Not More Than 78

The numbers in this arc fall and rise:

State	Technical Run	Result
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 local state	`46f3e0c`	132 passed
UP-018 DIX–ROBA	`b03b447`	141 passed
UP-019 Capability-Registry	`f11a60b`	147 passed
RV-001 correction	`9f14223`	148 passed

This table is **not** a progress curve. B-009 had fewer tests because large historical areas were moved to `unstable` or removed. UP-013 had fewer tests because it replaced a mandatory architecture along with its test suite. The numbers are meaningful only within the respective claim and source revision. A monotonically growing test counter could even have rewarded false stability here. [E-050]

The better question is:

> Which semantics does this suite cover, and which removed semantics is it specifically **not** supposed to preserve anymore?

6.8 B-014 — After Dismantling, Use Something Again First

The first thing built on the cleaned-up core was deliberately concrete. B-014 took two independent Applications, a specialized Typer Composition, and a small manual assembly Application. Using the existing bootstrap, this produced an executable multi-command CLI:

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

Typer remained optional. Function, group, and parameter IDs were preserved exactly. Parameters were named-only; real Python annotations remained the local source of types. Selection, renaming, or adaptation happened through normal wrappers, not through a second CLI policy language.

The E2E found a real bootstrap bug: the first multi-module launcher contained `),,` because of two separators. The previous single-module seed had not made this visible. The repair remained outside the core. Source and wheel then ran with 100 green tests; `src/dix/core` remained byte-for-byte unchanged. [E-040]

This is the constructive half of a replacement: after removing the wrong universality, the smaller core must support real work again. Otherwise, "less core" is merely an aesthetic claim.

6.9 B-015 — The Process Discovers Its Own Drift

While the core became cleaner, the operations process itself had silently grown weaker. B-015 compared the artifacts rather than just their names:

```
Block exists
=/derivation is append-only and traceable

testlaeufe/run-001 exists
=/run was executed and is reproducible
```

B-008 was the last block with a consistently maintained derivation. B-009 was the last run with a builder, pinned source snapshot, raw outputs, and provenance; later run directories were partly just editorial reports. The visible process had intended to retain its semantics and had retained only its folder names. [E-041]

B-015 therefore introduced a forward cutover:

- root `AGENTS.md` as a router;
- separate instructions for architecture block, implementation package, tickets, and test run;
- block with the current target state **and** an ongoing derivation;
- explicit slice approval before the UP;
- committed UP before committed tickets;
- builder plus immutable `run-NNN` with source, flow, commands, raw outputs, and exit codes;
- human acceptance as a separate status;
- no historical backfill.

The last point is crucial. The weaker old runs were not retroactively rewritten to fit a process standard that had not existed at the time. The drift itself remained evidence. B-015 was accepted.

6.10 B-016 — Config Gets Smaller by Becoming State

The next domain-level question began as config. A first slice, UP-016, created a real Pydantic model type from an owner-controlled declarative mapping spec. Pydantic remained an optional first-party module, not DIX core. The resolver accepted no values, sources, merge rules, or persistence. On `b2d5d14`, 118 tests ran.

The pressure then did not lead toward a larger config system. The reusable mechanics were more general and smaller at the same time:

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

dix/state/local
  one owner-relative model file
  + optional initial value
  → get() / set(complete_mapping)
```

UP-017 replaced `dix/config` with `dix/state` without a compatibility facade. `get()` returns a detached dump. `set()` validates a complete candidate, adopts it atomically within this local instance, and reports only whether the normalized value has changed. On ValidationError, the old value is retained.
Deliberately absent:

- actions, events, and callbacks;
- partial updates and path syntax;
- sources, merge, and priorities;
- registry, caches, and process sharing;
- file persistence;
- special Config, Typer, Exec, or ROBA semantics.

An owner can transparently export `get` and `set` or wrap them with normal Composition code and react to `changed` there. Multiple state instances are created through normal Composition aliases, not through a global state registry. The run on `46f3e0c` demonstrated 32 targeted tests, 132 tests in total, source and wheel, and the absence of the old config surface. UP-017 and then B-016 received human acceptance. [E-042]
This decision does not contradict the criticism of universal state. Here, DIX defines neither “workspace” nor “editor” nor a shared domain-level truth. It offers a concrete owner a small local, model-validated value boundary. Whether that becomes config, CLI state, or something else is decided by the consumer building on it.

6.11 The Status at the End of the Second Arc

Block / Slice	Substantive State at the Snapshot
B-007	implemented; operations artifacts still untracked, pinned separately
B-008	human acceptance granted; later replaced for the core by B-013
B-009	technically implemented; no human acceptance
B-010	agreed upon, but not implemented as a separate target state
B-011	accepted; minimal seed retained
B-012 / UP-012	block open; technical pressure test green, no architectural acceptance
B-013 / UP-013	replacement technically implemented; human acceptance of the block open
B-014 / UP-014	technically implemented; semantic acceptance open
B-015	operations cutover received human acceptance
B-016 / UP-017	local modeled state received human acceptance
B-017 / UP-018 / UP-019	integration technically far advanced; overall/slice human acceptances open
RV-001	technically implemented; human acceptance open

This coexistence is not an administrative error. An earlier accepted block may have been valid as a historical decision and have been replaced for today’s core. A later technically green slice may still be waiting for human acceptance. An open overall block may already contain explicitly approved and implemented subslices. Status therefore always needs an object and a point in time. [E-048, E-049]

6.12 What This Arc Adds to the First

The lifecycle case showed: green tests can precisely prove wrong universal semantics. The Norn case sharpens the statement:

1. The architecture can be small, elegant, locally isolated, and have received human acceptance.
2. A subsequent expansion can make it ever more technically consistent.
3. Only a different counterexample may reveal that the shared mandatory layer itself was the wrong ownership claim.

4. Local repair modes can conceal the error instead of solving it.
5. Replacement may preserve the valuable parts — contracts at real boundaries, Datamodel as a capability, isolated graphs — separately.
6. The first thing built afterward must prove in practice that the smaller core still holds up.

And B-015 adds a second level: the same rules apply to the development process. A builder, a ticket, or an `AGENTS.md` is not a magical truth machine. Evidence tools, too, need scope, failure paths, review, and a visible correction history.

The next section on DIX and ROBA is therefore no longer a futuristic integration sketch. It shows what happened when the reduced DIX core encountered a real external public API, one-time tokens, local capability sockets, and cross-process rebind.

7 Transferable Architecture Principles

7.1 Principles Are Not Universal Laws

The following principles were abstracted from DIX and ROBA. They are decision heuristics with counterpositions, not provable laws. Each principle therefore states the conditions under which it applies and its typical misuse.

7.2 1. Behavior Contract Before a Universal State Model

When heterogeneous systems are to cooperate, the discussion often begins with shared data: Buffer, Window, Workspace, Task, Session. This turns the first system's ontology into the norm.
A function contract starts more narrowly:

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

Each provider owns its native state and implements only the capability it can actually guarantee. Shared state is added only when it is itself a domain-level source of truth.
Applies especially when tools have different internal models.
Counterposition: For truly collaborative domains — such as a shared document, order, or lease — authoritative shared state is central. In that case, function-only would avoid necessary consistency semantics.
Misuse: Hiding every data query behind a function even though multiple consumers need the same versioned data structure.

7.3 2. Native Capability Before a Compatibility Interface

bwrap, Docker, and Windows process isolation differ in namespace, mount, credential, and lifecycle semantics. An early shared `Sandbox.start()` interface has two bad options:

- offer only the lowest common denominator;
- funnel target-specific options through generic flags.

Native modules may own their full semantics. When a concrete consumer needs a shared subset, a separate compatibility artifact emerges:

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

Applies especially when security or failure guarantees depend on the target.
Counterposition: For a narrowly defined organization, a deliberately limited shared interface can offer considerable benefits in operation and testing.
Misuse: Using "native" as an excuse to avoid any integration or shared terminology.

7.4 3. Mechanism and Policy May Grow Separately

ROBA Core can manage contexts, state, instances, and ACL principals. It does not know what "private profile," "company profile," "workspace," or "Architecture Mode" means. This separation does not devalue personal policy. It is precisely what allows that policy to be developed as a strong artifact in its own right.
Mechanism answers: what can be done reliably?
Policy answers: when, for whom, and with what meaning is it done?
Counterposition: Some policy is a systemic security invariant and must not be optional in the user layer. Access control itself is a mechanism; the minimum requirement "never put production secrets in untrusted sandboxes" can be binding organizational policy.

7.5 4. Core Versus Module Is a Question of Ownership

"The core should be small" is too superficial. A capability belongs in the core if:

- every layer above it depends on exactly the same meaning;
- the core can fully control the guarantee;
- building it externally would merely duplicate the same mechanism;
- the semantics are not tied to a target host or workflow.

It is more likely to belong in a module if:

- different targets have legitimate incompatible meanings;
- it carries an optional technology dependency;
- it is domain-level behavior rather than structural mechanics;
- several alternative policies are possible.

DIX Core owns structural graph lifetime. The Typer binding is a module. ROBA Core owns ACL and state primitives. Hub recovery is an external consumer.

7.6 5. Authority Applies per Boundary, Not Globally

A function signature in Python and a copy in TOML create two sources:

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

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

Even a new keyword parameter creates drift. For local in-process functions, the code can therefore be the signature authority; declarative specs then describe selection, aliases, dependencies, and exposure, rather than a second behavior language.

The second DIX arc, however, shows the necessary refinement. B-009 made a declarative contract the general authority and let it decide, through data model compatibility, whether concrete values were admissible. That was too strong for the local generator boundary: a rich, valid object was rejected simply because it could not fit losslessly into a smaller observation model. B-013 removed this global admission gate again. [E-036–E-039]

The rule is therefore not "code always wins," but:

> **INVARIANT** — The authoritative contract belongs to precisely the boundary whose stability must be guaranteed to its consumers. An observation or projection capability must not incidentally become global admission policy.

Counterposition: In cross-language or remote protocols, a machine-readable IDL contract must be authoritative. Code is then generated or validated against the IDL.

Misuse: Claiming "code is truth" even though external consumers cannot inspect the code and need stable versioning.

7.7 6. Declarative Specs Without a Behavior DSL

Declaration is strong for structure:

- which dependencies are used?
- which function is exposed?
- which config values apply?
- which module roots are trusted?

As soon as TOML expresses loops, conditions, error handling, and the order of side effects, a second programming language emerges. DIX keeps behavior in ordinary Python code and specs as an inspectable contract.

Counterposition: A limited DSL can be safer, more analyzable, and accessible to non-developers in highly standardized domains.

Diagnostic question: Is the DSL's semantics itself the product — or are we merely hiding code in YAML/TOML?

7.8 7. Explicit Wrappers Mark Ownership

Wrappers make sense when a consumer deliberately adopts someone else's function into its own API. They provide:

- a local name;
- a local signature;
- an adaptation point;
- error translation;
- a deprecation/migration boundary;
- inspectable provenance.

A generator can produce wrapper scaffolding, but cannot replace the local decision. Automatically re-exporting everything would turn a dependency back into an implicit API.

7.9 8. Local Dependency Graphs and Instance Isolation

Global registries are convenient until parallel configurations, tests, or tenants come into play. DIX distinguishes:

- Definition: static contract;
- Instance: concrete local graph;
- Owner scope: lifetime and visibility boundary.

This separation allows two instances of the same definition without state collisions. Shared capabilities remain possible, but must be explicitly set as runtime-scoped.

Counterposition: Some resources should be singletons. In that case, the singleton contract itself should be explicit, rather than an accidental consequence of a global registry.

7.10 9. Authoritative State Instead of Synchronized Copies

When Neovim, Robac, and a web client need the same context state, there are two models:

1. Each holds a copy and synchronizes events.
2. One service owns the authoritative state; clients read and write through its API.

ROBA implements model 2 for context/instance state deliberately held there. This reduces sources of conflict, but does not automatically solve offline operation, caching, availability, or persistence. In the current ROBA contract, this state is ephemeral: its authority applies within the running runtime, not across the loss of a daemon or machine.

Invariant: A DIX-ROBA module must not silently mirror state in DIX. An output may be a snapshot/descriptor, but authority remains with ROBA.

7.11 10. Three Authorization Levels Remain Separate

In the DIX-ROBA integration, which has now been technically implemented, three checks may apply:

1. **DIX Session:** May this participant use the specific interface surface?
2. **Function Exposure:** Is the function published in this application at all?
3. **ROBA ACL:** May the principal in use read, write, manage, or assign grants for the target resource?

These levels are not redundant:

```
Session permits interface
  =/Function is exposed
  =/Backend principal has resource permission
```

An all-encompassing "authenticated" Boolean would destroy error diagnosis and least privilege.

7.12 11. A Trust Boundary Is Not a Contract

DIX loads Python code only from trusted module roots. That is an admission decision. `composition.toml` then describes structure and published functions. It does not sandbox the code.

ROBA validates inputs both in the client and again at the daemon. The second validation is a trust boundary; the first improves UX and prevents unnecessary transport.

Diagnostic question: Which assumption protects against malicious or faulty code, and which merely describes expected cooperation?

7.13 12. Structural Lifecycle Versus Domain-Level Behavior

The B-005/B-006 case provides the clear formula:

- Core may manage its own definitions, instances, references, and graphs.
- Domain may define activation, readiness, retry, shutdown, and external resources.

A specialized service host module can offer a rich lifecycle protocol. The general graph core must not execute it implicitly.

7.14 13. Standards as Executable Artifacts

A standard such as “CLI inputs are named-only and env names explicit” can exist in three forms:

1. as a reminder in the prompt;
2. as a textual convention in the handbook;
3. as a CLI module that supports generation, validation, and testing.

The third form is the most robust. Text remains necessary to explain the reasoning. A prompt remains useful for steering a run. But the executable artifact makes deviations objectively visible.

Counterposition: Not every rule justifies tooling. Small, rare, or still-exploratory rules should not be prematurely cast into generators.

7.15 14. Honest Unsupported Semantics

Portability layers tend to conceal missing capabilities with empty values, no-ops, or best effort. This creates nominal compatibility, but unreliable meaning.

A more honest approach is:

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

An editor without `list_open_files` must not offer that capability. A Windows isolation module does not have to simulate a Unix socket. A compatibility artifact can deliberately define a smaller contract and fail hard for unsupported cases.

7.16 15. Replacement Before an Options Matrix

When a fundamental meaning is wrong, a compatibility flag seems friendly:

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

Yet every flag preserves a second architectural path. Before public stabilization, a clear replacement is often cheaper and easier to understand.

Counterposition: With external users in production, migration may be more important than internal purity. That calls for a versioned old contract, a deprecation path, and a measurable retirement point — not a silent break.

7.17 16. Negative Evidence Is a Result

A pressure test can show that:

- a capability is at the wrong layer;
- an ontology describes only one consumer;
- a rollback promise cannot control external side effects;
- a compatibility interface obscures security semantics;
- a supposedly generic feature has no second real consumer at all.

Such results must remain in the evidence chain. Anyone who documents only confirmed decisions produces architecture theater.

7.18 17. Composition Does Not Replace Responsibility

DIX makes it easy to combine functions and graphs. That does not legitimize every combination. An application contract must still explain:

- who owns the result;
- what side effects occur;
- which principal is used;
- how errors are visible;
- what repeated invocation means;
- who is responsible for destruction and external cleanup.

Composition is a mechanism. Responsibility remains architectural work.

7.19 18. Optional Local State Is Not a Global Ontology

After the criticism of universal shared state, the simple reaction would be to ban state from DIX entirely. B-016 shows the more precise boundary:

```
declared Pydantic model
→ exactly one composition-local model instance
→ get() as a complete mapping
→ set(full_mapping) as a validating replacement
```

The capability defines neither workspace nor editor nor shared persistence. It merely gives a concrete owner a modeled local value boundary. Two composition instances remain isolated. Anyone who needs persistence, partial updates, events, or distributed consistency must add those semantics as a separate artifact. [E-042]

Counterposition: Full-model replacement can be inefficient and prone to conflicts with large or concurrent models. In that case, a different state contract is needed — not a gradually expanding `set()`.

7.20 19. Passing Functional Tests Do Not Validate an Architecture Boundary

A suite can prove that the end system does what its test model demands, yet still conceal that:

- ticket A already contains ticket B's responsibility;
- a commit can no longer be reviewed in isolation;
- cleanup models only the case of an unambiguous failure;
- an operations run was built from the wrong source surface.

That is why a risky slice needs a review after the feature run whose questions are not simply copied from the same acceptance criteria. RV-001 found errors at exactly this level even though all feature tests passed. [E-045]

7.21 20. "Create Failed" Is Not an Unambiguous State

External mutations have at least three relevant outcomes:

```
Resource not created
Resource created and success confirmed
Resource created, but confirmation lost
```

The third case arises from timeouts, interrupted transport, or a lost response. Rollback code must therefore not make cleanup depend exclusively on a received return value. It must mark the attempt before the mutation, attempt an idempotent compensation or one that can be safely classified, and suppress only errors that are demonstrably harmless.

Counterposition: Not every external API allows safe retries or idempotency. In that case, the state must explicitly be escalated as unknown; a local `false` would be invented certainty.

7.22 21. Evidence Is Itself Fallible and Therefore Append-Only

An artifact builder can quote shell code incorrectly. A supposed snapshot can lose Git provenance that the test expects. A staging check can fail without consequence if there is no fail-fast chain. Evidence is therefore subject to the same basic rules as product code:

- exact inputs and source commits;
- explicit exit codes;
- checksums and raw output;
- negative paths;
- no silent overwriting of a failed run;
- visible correction through a new run or commit.

An invalid run does not substantiate the domain-level claim. Its preserved failure cause does, however, substantiate that the process does not erase its own fallibility from history. [E-046, E-047]

7.23 A Compact Decision Matrix

Question	More likely core	More likely module/consumer	Warning sign
Who controls the guarantee?	Core fully	Target system/host	No one fully
How many legitimate semantics?	One	Several	Options matrix grows
Is state authoritative?	Yes, core domain	External service/tool	Several synchronized copies
Is behavior declarative?	Structural	Ordinary code	DSL grows unnoticed
Is trust clarified?	Narrow boundary	Native module	Config is mistaken for a sandbox
Is there a real second consumer?	Proven	Keep it specific for now	Hypothetical universality
What happens when unsupported?	Hard contract error	Native error/adapter	No-op or empty fake value
What does “authoritative” mean?	Within a named runtime	External durable source	Ephemerality is left unstated
Has a create unambiguously failed?	Proven on the server side	Escalate state as unknown	Cleanup based only on return value
Who checks the evidence?	Independent review path	Named manual acceptance	Builder declares itself valid

This matrix does not decide automatically. It merely forces the hidden reasoning to the surface.

8 DIX × ROBA: From Target Vision to Optional Vertical Slice

8.1 Status of This Chapter

> **VERIFIED / TECHNICALLY IMPLEMENTED / HUMAN ACCEPTANCE PENDING** — The dependency direction that was still hypothetical in V1 has now been implemented: The optional first-party module `dix/roba` consumes only the public Python API of the standalone package `roba`. Source, wheel, and cross-process runs are green. At the snapshot, the overall semantic premise of B-017 and UP-018, UP-019, and RV-001 had not yet received human acceptance. [E-043–E-048]

This is a deliberately unwieldy status line. It prevents three convenient but false shortcuts:

1. The integration is no longer merely a target vision.
2. Technical existence is not yet semantic acceptance.
3. A green E2E does not retroactively make an open review irrelevant.

8.2 What V1 Had Expected — and What Was Actually Built First

The previous edition proposed, as the smallest integration cycle, reading exactly one ROBA context value from a DIX application and writing an instance value with a minimal ACL. That was a plausible pressure test, but it was not the first slice chosen later.

The operations work started earlier in the chain. Before DIX consumes domain-level state from ROBA, it must first be able to configure, start, address, and stop an external runtime without hidden host policy. UP-018 therefore built this chain:

```
DIX Local State
→ explicit ROBA configuration
→ public ROBA Python API
→ Daemon start / status / stop
→ DIX Application
→ manually composed Typer CLI
```

This is not a defeat for the old hypothesis. It is its first contact with the actual build path. The smaller pressure point was not an arbitrary state value, but the runtime and configuration boundary. [E-043]

8.3 Two Systems Remain Two Systems

The integration holds up because it does not attempt a retrospective merger:

```
DIX
= local Composition/Application graphs
+ explicit Functions
+ optional first-party modules
+ target-system-specific delivery artifacts

ROBA
= standalone ephemeral context-state service
+ Contexts and Instances
+ TokenPrincipals and SocketPrincipals
+ Resource ACLs
+ Daemon, Control, and Context endpoints
```

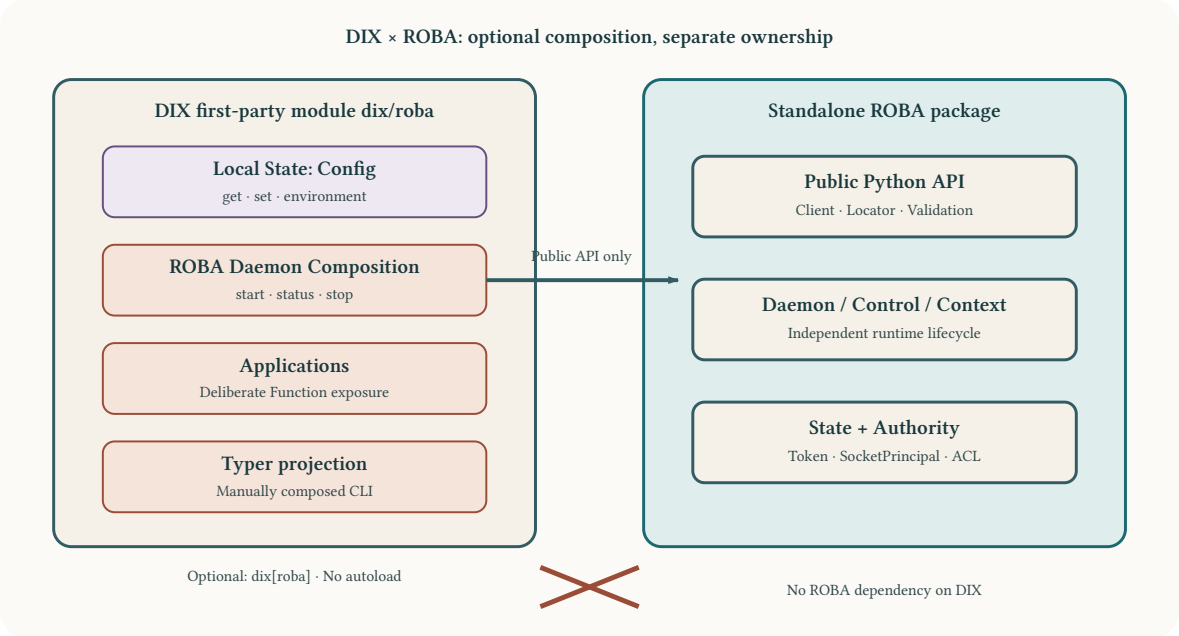

Figure 6: The implemented dependency direction: The optional DIX module composes local config, the ROBA public API, applications, and CLI. ROBA knows nothing about DIX.

The dependency invariant from V1 remained intact:

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

The packaging makes the boundary visible. DIX itself has no base dependency on ROBA. Only `dix[roba]` pulls in the independent package and the optional dependencies required for this module. The module does not autoload. A consumer loads it explicitly, just like any other DIX module. [E-043]

This direction matters more than the concrete CLI. If ROBA now had to know DIX types, DIX graphs, or DIX launchers, the result would not be an integration, but mutual assimilation.

8.4 UP-018 — The First Real Integration Seam

8.4.1 Invocation-Local Configuration Instead of an Inherited Environment

The `config` composition holds a complete local model for exactly one ROBA daemon configuration. It exposes three narrow functions:

```
get()                → complete configuration
set(full_mapping)    → validate and replace completely
environment()        → explicit process environment
```

It reads neither an implicit `roba.toml` nor inherited `ROBA_*` variables. When building the process environment, it even removes existing `ROBA_*` values and sets only the locally modeled values afresh.

This means the configuration is not global "DIX state." It belongs to a specific composition instance. Two DIX instances can address two ROBA runtimes without accidentally sharing their configuration state.

8.4.2 Adapter Through the Public Python API

The `daemon` composition uses ROBA neither through its CLI nor through internal socket details. It adapts the public Python API for exactly three operations:

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

The DIX application above it exposes these functions deliberately. The Typer application, in turn, projects them into a scriptable CLI. Each layer remains a normal consumer of the preceding one:

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

There is no new universal runner in the core. Nor was the CLI magically generated from a global contract system. It is manually composed from concrete applications — exactly as the reduced B-013 core provides for.

8.4.3 What the Run Demonstrates

UP-018 checked:

- optional packaging through source and wheel;
- instance isolation of the configuration;
- removal of inherited ROBA variables;
- start, status, and stop through the ROBA public API;
- application and Typer projection;
- error paths and cleanup;
- unchanged ROBA core tests.

The technical target revision `b03b447` ran green with 141 DIX tests and 59 ROBA tests and 32 ROBA subtests. In the archived outputs, tokens were printed only as fingerprints, not as secrets. This substantiates the slice; it substantiates neither production security nor its human acceptance. [E-043]

8.5 UP-019 — From Daemon Operation to an Ephemeral Capability Registry

After the first slice, a concrete pressure point involving long-running processes remained open: A process A can create a ROBA daemon and context, terminate, and later be replaced by process B. If all owner and control credentials live only in the first process, the technically running daemon is practically no longer manageable. UP-019 does not solve this with a persistent secret database. It builds an ephemeral registry inside the running ROBA daemon using the ROBA primitives already available.

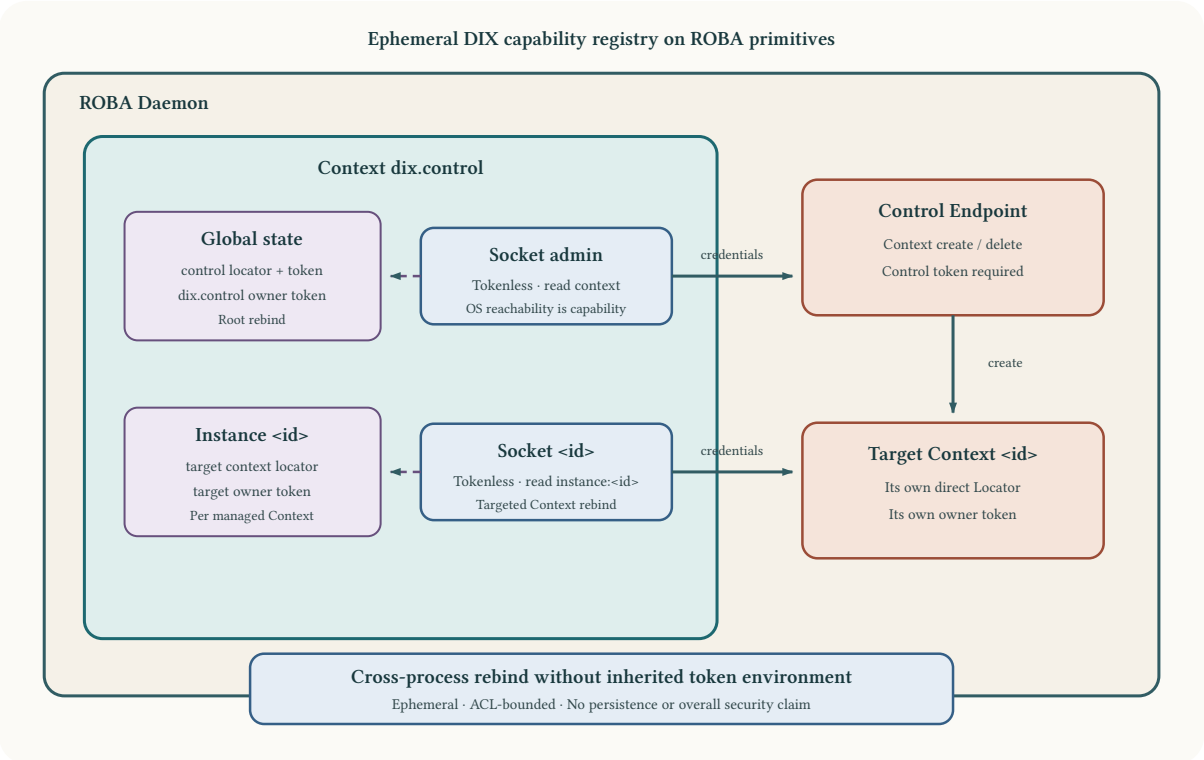

Figure 7: The DIX-owned capability registry: A control context holds root and context credentials; tokenless sockets restricted by ACL allow targeted rebind.

8.5.1 Topology

Bootstrap creates:

```
ROBA Daemon
└── Context dix.control
    ├── global state
    |   ├── control locator
    |   ├── control token
    |   └── owner token of dix.control
    ├── Socket admin
    |   └── read on context state
    └── per managed context <id>
        ├── Instance <id>
        |   ├── direct context locator
        |   └── owner token of the target context
        └── Socket <id>
            └── read exclusively on Instance <id>
```

The root socket and the manager sockets are SocketPrincipals. Anyone who can reach the respective Unix socket needs no additional token for this limited operation. The capability lies in the reachability of the socket plus its server-side ACL.

This is deliberately not a claim that every local socket is secure. OS permissions, mounts, namespace boundaries, and socket distribution remain external authority decisions. ROBA enforces only the resource ACL represented by the socket.

8.5.2 Four Visible Functions

The control composition exposes four central operations:

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

`bootstrap()` starts a fresh daemon, creates `dix.control`, writes the root credentials into its global state, and creates the read-only admin socket.

`control_credentials()` reads these credentials back through the tokenless admin socket. A later process therefore does not need to have inherited the one-time bootstrap result in its environment.

`create_context()` uses the recovered control authority, creates a target context, and stores its locator and owner token in a registry instance of the same name. It then creates a manager socket that may read only this instance.

`context_credentials()` rebinds exclusively through this context-specific socket. The consumer thereby gains access to exactly the registered target without also being able to read the root registry.

8.5.3 Why This Is Not State Mirroring in the Old Sense

The registry holds credentials and locators as explicit operational state. It does not continuously mirror the domain-level state of the target contexts. There is no sync loop and no claim to be a second truth about their contents.

An important limitation is nevertheless necessary: The registry is authoritative only as long as the daemon lives and the entries are not changed outside its management path. It provides no durability guarantee. “Authoritative” here means **for this running capability rebind**, not “persistent through loss of the daemon process or the machine.”

8.5.4 What the Run Demonstrates

The technical revision `f11a60b` passed ten focused registry tests, 147 DIX tests, and a real cross-process E2E. Process A created the daemon, registry, and context; a separate process B recovered credentials through the capability sockets. ROBA remained green at its referenced commit with 59 tests and 32 subtests. [E-044]

8.6 RV-001 — Why 147 Green Tests Were Not the End

The independent review checked not only function output, but ticket boundaries, rollback, and operations evidence. It found three different kinds of problem.

8.6.1 1. A Ticket Boundary Had Been Violated

DX-072 was supposed to build the root registry. The commit, however, already contained substantial managed-context functionality from DX-073. The final system could work; the claimed commit evidence was nevertheless sliced incorrectly.

This is not a matter of style. If a commit carries more architectural responsibility than its ticket, neither regression nor review can later establish unambiguously which change proves which claim.

8.6.2 2. A Rollback Path Was Incomplete

When creating a capability socket, the cleanup marker was set only after `socket_create()` returned successfully. This left a crucial intermediate state unmodeled:

```
Server has created socket
→ Response does not reach client
→ Call raises error
→ Local code believes: socket was never created
→ Rollback leaves resource behind
```

The fix marks the attempt as begun **before** the call. On error, deletion is attempted; only a demonstrable `not found` may be ignored. Other cleanup errors remain visible. [E-045]

8.6.3 3. The Evidence Process Itself Produced Errors

The first review run was invalid because an unquoted heredoc interpreted backticks as shell commands, corrupting metadata. The run was preserved with `FAILED.md` rather than overwritten. Run 002 then substantiated five review tests, 148 DIX tests, and the unchanged ROBA suites.

When this evidence was first committed, a hidden staging directory also ended up in the commit. The cause was mundane and instructive: `git diff --cached --check` was not chained to the commit with fail-fast behavior. A follow-up commit removed the area again instead of smoothing out history. A third run then reran that exact corrected builder against the same pinned DIX and ROBA commits. All ten exit artifacts read `0`; no staging directory remained. [E-046]

8.7 What Is Confirmed, Corrected, and Open

8.7.1 Confirmed

- DIX can consume ROBA optionally and exclusively through its public API.
- Configuration can remain composition-local and free of inherited ROBA policy.
- Daemon lifecycle can be managed through normal DIX compositions, applications, and Typer projection.
- An ephemeral capability registry can be built with ROBA state, instances, and SocketPrincipals without an additional ROBA core mechanism.
- Cross-process rebind works technically.

8.7.2 Corrected

- The first relevant slice was runtime configuration and daemon operation, not arbitrarily reading and writing a domain-level value.
- “A run exists” is not enough; the run itself can be invalid.
- “All tests green” is not enough; ticket boundaries and ambiguous cleanup states need pressure of their own.

8.7.3 Open

- At the snapshot, B-017, UP-018, UP-019, and RV-001 await human acceptance. [E-048]
- The planned native bwrap E2E with a specifically mounted SocketPrincipal is not part of this revision.
- Security audit, remote transport, Windows-native authority, and persistent recovery remain outside the claim.

- In one place, the DIX README still contains older future-tense language about ROBA integration; code, module README, and tests show the newer state.

8.8 The Still-Open Sandbox Pressure Test

The strongest next native slice remains similar to V1, but is now built on real registry evidence:

1. The parent possesses root or context authority.
2. It creates a SocketPrincipal with a minimal ACL.
3. bwrap receives only the project CWD and this socket.
4. The child possesses no owner/control token.
5. A DIX application in the child executes permitted operations.
6. Access outside the ACL fails distinguishably.
7. Socket revocation and cleanup are observed under real error paths.

Linux-bwrap, Docker/OCI, and Windows may still have their own native modules for this. A later small compatibility function can project common cases. But it is the result of actual commonality, not a prerequisite for integration.

8.9 What This Vertical Slice Proves Methodologically

V1 could only argue that two public APIs fit together conceptually. V1.1 now has stronger, but still limited, evidence:

```
Hypothesis
→ optional packaging
→ local config owner
→ public API adapter
→ Application + CLI
→ ephemeral capability registry
→ cross-process rebind
→ independent rollback/boundary review
```

The departure from the old first slice is precisely the gain. ACDD does not require an early target design to remain right. It requires us to make visible which real pressure changed it, what now exists technically, and which meaning still awaits acceptance.

9 AI Cooperation, the Company, and Public Communication

9.1 AI Is an Accelerator and an Error Amplifier

AI can process large amounts of context in an architecture cycle, formulate alternatives, implement code, and assemble evidence. It can just as quickly carry a false abstraction consistently through the entire system. That is why AI needs clear authority and evidence boundaries, not a mystical special role.

9.1.1 The Human Is Responsible For

- intended meaning and priority;
- legitimate organizational policy;
- risk acceptance;
- decisions at genuine semantic forks;
- final manual acceptance;
- approval of internal or public claims.

9.1.2 AI Can Responsibly Carry Out

- examining the existing system and Git history;
- finding contradictory contracts;
- making a strong case for opposing positions;
- checking target designs for completeness;
- deriving implementation packages and tickets from confirmed semantics;
- implementing and committing small tickets;
- running tests and isolated artifact runs;
- maintaining evidence/claims ledgers;
- auditing documentation against code and test output.

9.1.3 AI Must Not Silently Decide

- whether new behavior belongs in Core or Module when ownership is open;
- whether a replacement may break backwards compatibility;
- whether a security boundary is acceptable;
- whether personal policy becomes a general standard;
- whether missing acceptance counts as passed;
- whether a new public assertion follows from indirect evidence.

9.2 The Goal-Prompt Is an Execution Contract, Not Truth

A good goal-prompt is detailed because it fixes the boundary, scope, ticket sequence, tests, and completion audit. Its length gives it no semantic authority. The chain of authority remains:

```
confirmed architecture state
→ implementation package
→ goal-prompt
→ code and evidence
```

If an agent discovers a contradiction between the prompt and the block, it must stop or return to the block. “Run autonomously through to M1” applies to implementation work within clarified semantics, not to secretly inventing those semantics.

9.3 Prompt Engineering Does Not Replace a Standard

A repeated instruction such as

> “Always use named parameters and explain all options inline”

is useful, but weak:

- it can be forgotten or shortened;
- models interpret it differently;
- it is not automatically regression-testable;
- deviations only become visible in review.

An executable standard can express the same rule in a CLI descriptor, generator, validator, and test. B-006 uses precisely this idea for an opinionated Typer module. The prompt then only needs to require use of the standard artifact and verification of its output. [E-015, E-016]

> **COUNTERPOSITION** — Rules that are still unstable do not belong in tooling immediately. A prompt or block is cheaper for exploration. Only repeated, understood semantics should become an executable standard.

9.4 Inspectability Instead of Trust in the Agent

An AI-assisted run should be reconstructible from the outside:

- which sources and commit states were used?
- which target design was binding?
- which files changed?
- which commits carry which increment?
- which tests ran with which stdout?
- which isolated artifact was produced?
- what was checked manually?
- which open issues remained?

Transcripts can supplement this, but they make poor primary documentation. They are long, contain intermediate assumptions, and mix questions, interpretations, and decisions. The block distills the semantic development; Git and tests provide evidence of the technical development.

9.5 The Process Must Remain Understandable Without AI

If only the executing model can explain how a cycle works, the process is not inspectable. A human must be able to use the artifacts to:

1. understand the initial pressure;
2. follow the alternatives;
3. identify the chosen contract;
4. find code and test evidence;
5. check acceptance status;
6. recognize the next open pressure.

AI increases throughput. It does not replace semantic readability.

9.6 Case Study: The Small Assignment and the Perfect Platform

The initial pressure behind DIX was not a platform mandate. A small internal application needed a robust path from form-oriented inputs to actions. An older proof of concept could serve as a source of experience; the new codebase was to be cleaner, independent, and possibly reusable later. This description is deliberately anonymized. Neither the client nor the domain-specific form data matters to the architecture question. [S-15]

> **WORKSHOP SCENE · ANONYMIZED RECONSTRUCTION** S-15 · PARAPHRASE
>
> The actual delivery was small enough that it could have been handled in a local script. At the same time, the known friction was substantial enough to immediately bring renderers, actions, config, states, reuse, and multiple interfaces to mind. This is precisely where architecture work can either disappear or swallow the assignment.

9.6.1 Danger One: The Assignment Becomes a Pretext for a Platform

A technically ambitious team often recognizes recurring problems before the organization is ready to support them as a product. That can be a strength. It becomes a trap when the concrete assignment is silently expected to pay for research, generality, and future users.

Typical warning signs:

- The first real consumer is waiting for a generic plugin model.
- Nonexistent second backends are already determining the Core.
- “Reusable later” replaces a named Product Owner.

- Acceptance criteria check platform mechanics instead of the value delivered.
- every local peculiarity is included as an option in the general contract;
- scope growth is presented as an inevitable part of architectural quality.

An elegant platform that delivers the small assignment too late or with too much risk is no architectural gain. It is a different undertaking without an honest budget decision.

9.6.2 Danger Two: Recurring Friction Is Paid for Locally Every Time

The opposite extreme looks commercially sober: solve only the assignment, abstract nothing. But if every new tool reinvents option parsers, error formats, config resolution, permissions, renderer binding, or instance isolation, the organization pays for the same insight several times. The costs simply do not appear as a platform project. They are spread across integration errors, incompatible operation, maintenance, and later migration.

The small `argparse` example from B-006 is ideal here. For a single tool, the choice was quick and correct. Only alongside many possible tools did it become apparent that named parameters, environment, help text, JSON, error channel, and exit codes would be decided anew every time. It does not follow that "every CLI must use DIX." What follows is: if this exact contract pressure recurs, it may have product value of its own. [S-13]

9.6.3 Three Tracks Instead of One Mixed Roadmap

The practically viable separation is:

```
DELIVERY
  What must work reliably for this assignment?

EVIDENCE
  Which friction or invariant might apply beyond this assignment?

PRODUCTIZATION
  Which deliberately funded, maintained, and published capability
  do we actually extract?
```

The tracks are connected, but they do not have equal priority in day-to-day work. Delivery has an assignment, a deadline, risk, and a concrete acceptance process. Evidence initially has only the duty to preserve observations in a small, lossless form. Productization begins only through a separate decision.

9.6.3.1 1. Delivery

For the initial form-oriented case, the delivery contract might have read something like this:

- accept and validate defined inputs;
- execute precisely named actions;
- make errors visible to the real consumer;
- test the chosen browser/server path E2E;
- promise no general workflow engine.

B-001 did something similar: config, spec, runtime state, HTTP, and HTML were connected as a small, complete slice. The UI-oriented Core was not yet the later ontology. It was still allowed to exist for this cycle because it delivered a real path and could undergo acceptance. [E-001]

Delivery protects against the sentence: "We can only deliver once the perfect abstraction is finished."

9.6.3.2 2. Evidence

During delivery, recurring pressure points do not immediately become a library, but verifiable notes:

```
Observation:
  The HTML renderer owns template and interaction semantics.

Recurrence indicator:
  A second interface would represent the same semantics differently.

Risk:
  If we move widgets into Core, presentation binds the domain capability.

Next cheap test:
  Separate renderer ownership in a bounded slice.
```

```
Status:
  EVIDENCE, not yet a product promise.
```

Evidence may live in the assignment's repository, in an architecture block, or in a company-wide register. It does not yet need a general API. Its value is that the next team does not have to start again from a vague memory. Evidence protects against the sentence: "We have had this three times already, but nobody remembers exactly why it hurt."

9.6.3.3 3. Productization

A reusable capability receives a separate decision when at least one substantive justification exists:

- multiple real consumers generate the same semantic pressure;
- a strategic platform decision deliberately funds upfront work;
- security, compliance, or operations require a uniform contract;
- the integration effort is measurably greater than maintaining a shared artifact;
- a vertical pressure test shows that the shared semantics are actually smaller and more stable than the native systems.

Productization then needs its own ownership:

- Product/Capability Owner;
- support and version boundary;
- trust and distribution model;
- compatibility decision;
- independent tests and artifact runs;
- migration or deliberately no compatibility;
- a budget that is not secretly buried in the original assignment.

Productization protects against the sentence: "Because the code is already here, it is now automatically our platform."

9.6.4 One Artifact, Three Status Values

The same code can pass through the tracks in sequence without rewriting its earlier meaning:

Point in time	Delivery	Evidence	Productization
first slice	concrete task works	initial friction documented	not decided
second pressure test	existing assignment remains stable	shared contract is put under pressure	exploration funded
extracted module	consumer migrates deliberately	origins and boundaries remain visible	owner, version, and support established

Conversely, an artifact created locally may remain local permanently. Not every good idea has to become a product. A strategic platform may also begin before the second consumer — but then the honest justification is "we are investing deliberately," not "the small assignment required it."

9.6.5 A Concrete Company Flow

A lightweight company building block can look like this:

```
auftrag/
├── delivery.md          goal, scope, acceptance, deadline
├── architecture.md      only the contracts needed for delivery
└── evidence.md          possible recurring pressure points

capabilities/
└── candidate-index.md   organization-wide, as yet nonbinding evidence

productization/
└── <capability>/         separate block, owner, budget, and lifecycle
```

`evidence.md` may contain three lines. The index may reject a capability again. Only the productization decision creates an obligation. This makes architecture observations available for learning across the organization without turning every customer problem into a pretext for the "perfect platform."

> **COUNTERPOSITION** — In small teams, additional tracks can create more maintenance than benefit. In that case, status markers in one file are enough. What matters is the semantic separation, not the number of folders.

9.6.6 What DIX Proves in This Case Study — and What It Does Not

For the first book snapshot, the evidence shows that six short cycles extracted several responsibilities from the concrete slice and rejected or replaced two sets of semantics. V1.1 adds an even harder finding: the subsequently accepted Norn/Strand architecture was actually developed further and replaced again under a generator counterexample. Local State and an optional DIX×ROBA seam were then built on the smaller Core. This does not establish that DIX is already the economically right platform for a company, that its maintenance costs are lower, or that other teams need the same contract.

The correct public statement is therefore not "our universal platform emerged from a small assignment." It is: the assignment generated a real pressure test; recurring evidence gave rise to an independent architecture experiment whose current technical state and open issues are verifiable.

9.7 Public Casebook and Company Handbook Are Different Products

A single document with `if public/if internal` switches quickly becomes unreadable and risky. A shared substantive foundation with two overlays makes more sense.

9.7.1 Shared Foundation

- ACDD terms;
- block/package/ticket/evidence contract;
- generic templates;
- public DIX architecture;
- reproducible, redacted cases;
- general roles and drift warnings.

9.7.2 Public Casebook

Goal: understandable communication of the substance without internal data.

Contains:

- generic DIX cases;
- public commit ranges;
- technical alternatives and replacements;
- reproducible artifact runs;
- redacted test outputs;
- clear snapshot and open-issue boundaries;
- no private raw Operations material, tokens, or company data.

A public case should not claim "ACDD guarantees better architecture." It should show: here was the assumption, here the slice, here the evidence, here the correction. Readers can evaluate the chain of evidence themselves.

9.7.3 Company Handbook

Goal: a binding internal engineering and approval process.

In addition to the foundation:

- roles and RACI;
- system/data classification;
- security and compliance evidence;
- internal repository and signature rules;
- Definition of Ready/Implemented/Accepted;
- review and approval boundaries;
- AI usage and data policies;
- internal case studies;
- incident/exception process;
- retention and deletion of transcripts/snapshots.

9.8 A Possible Company Model

9.8.1 Roles

Role	Decision	Evidence
Product/Domain Owner	intended value, domain-level acceptance	Use Case, Acceptance Notes
Architect	contract, invariants, replacement	Block, Decision, Review
Tech Lead	feasibility, ticket boundaries	Package, Plan
Developer/Agent	code within the contract	Commits, Tests
Security/Compliance	trust/data/control boundaries	Threat Model, Controls
Independent Reviewer	claims against evidence	review record

9.8.2 Approval Boundaries

- **Exploration:** no product commitment; WIP artifacts clearly marked.
- **Target Agreed:** implementable, not yet proven in reality.
- **Technically Implemented:** code/tests/run available, no user acceptance.
- **Accepted:** real acceptance and claims audit completed.
- **Public:** redaction, license, security, and claim boundary checked.

9.8.3 Security/Compliance Evidence

ACDD does not replace a security process. It can integrate that process's chain of evidence:

```
Threat / Requirement
→ architecture control
→ Implementation
→ automated security test
→ Deployment Evidence
→ manual/organizational approval
```

What matters is clarity of status. A unit test for ACL parsing is no evidence that production file permissions are set correctly. A successful penetration test is not domain-level acceptance.

9.9 Definitions for the Company

9.9.1 Definition of Ready

- semantic pressure and owner named;
- data/security class known;
- architecture block stable enough;
- non-goals and migration clarified;
- testable vertical slice;
- AI is allowed to see and edit the sources that the goal-prompt presupposes.

9.9.2 Definition of Accepted

- code and tests on a fixed commit;
- isolated artifact run;
- manual acceptance by an authorized role;
- security/compliance evidence where required;
- claims ledger without unmarked inference;
- rollout/rollback decision;
- public communication approved separately.

9.10 AI Usage Rules as a Company Overlay

1. **Capability logic:** visibility is authorization; high-value scopes are assessed separately.
2. **No hidden semantics:** return to the block at an architecture fork.
3. **No invented evidence:** record only actual stdout, commits, and acceptance.
4. **Snapshot discipline:** do not silently mix parallel source changes.
5. **Secret hygiene:** scan outputs and PDFs before publication.
6. **Inspectable Changes:** small, named commits; no cosmetic mixed changes.

7. **Human Acceptance:** an agent may report technical completion, but must not simulate user acceptance.
8. **Model Independence:** process artifacts must remain readable by another human or tool.

9.11 A Substantive Textbook from Real Operations

Operations is richer than a publishable book. It contains:

- rough questions;
- discarded intermediate states;
- private contexts;
- local paths and test fixtures;
- repetitions;
- WIP without a commit.

The editorial task is **not** to shorten everything until only a clean success story remains. It consists of four transformations:

1. **Distill:** extract the relevant architectural movement from the raw history.
2. **Verify:** check against commits, code, tests, and artifact runs.
3. **Classify:** fact, inference, interpretation, open, rejected, replaced.
4. **Redact:** remove private details without smoothing out the negative evidence.

The result is not a marketing case, but a substantive casebook.

9.12 An Honest Public Narrative

Bad:

> "In five days, AI helped create a revolutionary modular platform."

Better for the first snapshot:

> "Six closely documented cycles moved DIX from a UI-oriented foundation slice to a capability/composition/application core. Two central sets of semantics were deliberately rejected or replaced. The snapshot through DX-029 has 209 passing tests; the full B-006 E2E and external transferability remain open."

The second narrative is less spectacular and substantively much stronger. It invites scrutiny instead of demanding trust. V1.1 may add:

> "The subsequent cycle first implemented and accepted a composition-scoped contract core, replaced it again after a concrete generator counterexample, and built an optional ROBA integration on the reduced core. 148 DIX tests and the cross-process E2E have been reproduced; human B-017/package acceptance remains open."

This narrative is no proof of company suitability either: external application, costs/benefits, process comparison, maintenance observation, and security/compliance overlays remain **OPEN**. Appendix C elaborates on this claim boundary.

10 Handbook: Apply ACDD Directly

10.1 Minimal Starting Point

For the first cycle, a team does not immediately need a new tool. A repository folder is enough:

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

The folders are not dogma. What matters is the direction:

- while semantics remain open, truth lives in the block;
- after the target design, an implementation package is derived;
- tickets and commits form provable steps;
- the artifact run and acceptance close the cycle;
- claims reference evidence rather than just the final state.

10.2 Roles and Responsibility

Role	Responsible for	Must not delegate
Semantics owner	Meaning, goal, non-goals, acceptance	final semantic decision
Architecture partner	Counterpositions, boundaries, sharpening the target design	feigning consensus
Developer	Implementation, local design decisions within the contract	hidden semantic change
AI agent	Inspecting the existing state, structuring variants, implementing tickets, collecting evidence	silently deciding unclear forks
Reviewer	Contract/code/evidence consistency	acceptance without a real artifact
Operator/Security	Runtime, trust, and deployment boundaries	deriving domain semantics from infrastructure

In small teams, one person can fill several roles. The responsibilities should nevertheless remain recognizable in the artifact.

10.3 Template 1 — Architecture Block

```
---
id: B-XXX
title: <short problem name>
status: exploration | target-agreed | implemented | accepted | replaced
opened: YYYY-MM-DD
source: <previous block, incident, use case>
---

# B-XXX — <Title>
```

```
## Initial Pressure
What observable behavior, risk, or real goal gives rise to the block?

## Current Findings
- VERIFIED: ...
- INFERENCE: ...
- OPEN: ...

## Terms
What exactly do the disputed words mean in this block?

## Plausible Previous Model
Why was/is it reasonable? Do not build a straw man.

## Counterpositions and Alternatives
### A — ...
Benefits, costs, break.
### B — ...
Benefits, costs, break.

## Architecture Boundaries
- Owner:
- Source of Truth:
- Trust Boundary:
- Dependency Direction:
- Error/rollback claim:

## Current Target Design
Precise contract in a few rules.

## Invariants
- ...

## Non-Goals
- ...

## Falsification Conditions
What real finding would replace or sharpen the target design?

## Open Decisions
- O-001 ...

## History
### YYYY-MM-DD — <Correction>
What changed and why?

## Acceptance
Who checked which artifact with what result?
```

10.3.1 Block Quality Check

- Does the initial pressure describe a problem rather than a solution?
- Is the previous model presented fairly and plausibly?
- Are alternatives real contracts rather than buzzwords?
- Can someone name the source of truth and authority?
- Is it clear what is and is not guaranteed on error?
- Is there at least one falsifiable finding?
- Was a replacement marked as such?

10.4 Template 2 — Semantic Sharpening

These questions can be used in review or AI cooperation:

10.4.1 Responsibility

- Who owns the meaning of this term?

- Can the proposed layer fully control the guarantee?
- Is this mechanism, policy, UX, or domain behavior?
- Is a personal way of working being sold as general semantics?

10.4.2 State

- Is the state authoritative or a projection?
- Who may change it?
- What happens with parallel instances?
- Why does the consumer need data rather than a narrower function?
- What staleness or conflict semantics are being claimed?

10.4.3 Functions and Contracts

- Which function does a consumer really need to be able to call?
- Where does its authoritative signature come from?
- Is a wrapper local ownership or merely boilerplate?
- What does unsupported mean?
- Is a hook structural or domain-level?

10.4.4 Trust and Authority

- What is a credential, what is a locator, what is an identity?
- Which check protects which resource?
- Is declarative configuration wrongly presented as a security boundary?
- What can external code change outside the claimed rollback?

10.4.5 Future

- Is there a real second consumer?
- What "future flexibility" is currently only being assumed?
- Would a native target module be more honest?
- What compatibility artifact could be created separately later?

10.5 Template 3 — Counterposition / Alternative

```
### Alternative A — <Name>

**Strongest argument for it**
...

**Contract**
...

**Properties gained**
- ...

**Costs / properties lost**
- ...

**Real Pressure Test**
...

**Falsifying finding**
...

**Why currently adopted / rejected / open**
...
```

The requirement "strongest argument for it" prevents straw men. An alternative without a testable contract is merely a buzzword.

10.6 Template 4 — Implementation Package

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

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

# UP-XXX — <Title>

## Binding Architecture State
Brief reference to the confirmed block sections.

## Goal
What vertical slice is being created?

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

## Non-Goals
- ...

## Planned Ticket Slice Sequence
1. TX-001 — ...
2. TX-002 — ...

## Automated Acceptance
- Positive path
- Error path
- Isolation
- Teardown
- Public Contract

## Isolated Artifact Run
Inputs, build, runtime, outputs, review order.

## Manual Acceptance
Concrete actions and expected observation.

## Stop Conditions
- semantic fork;
- required hidden core expansion;
- non-reproducible evidence;
- source/trust boundary unclear.

## Goal-Prompt
Executable assignment; no new semantics.
```

10.7 Template 5 — Ticket

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

# TX-XXX — <provable increment>

## Contract Delta
Which observable statement changes after this ticket?

## Scope
- ...

## Non-Goals
- ...
```

```
## Implementation Notes
Permitted local degrees of freedom; no new architecture decisions.

## Tests
- ...

## Evidence Output
Which output/file/commit substantiates the result?

## Commit
Hash and subject after implementation.

## Open Acceptance
Only if the ticket itself requires manual checking.
```

10.7.1 Good and Bad Tickets

Bad	Better
“Change models”	“Remove reserved lifecycle fields from Composition/Application spec”
“Add tests”	“Demonstrate structural rollback on constructor error without a domain-level hook”
“Build CLI”	“Generate Typer binding from normalized option descriptors”
“Cleanup”	“Hard-reject module unload while an external instance is alive”

10.8 Template 6 — Goal-Prompt

```
# Goal: Implement UP-XXX Through to Technical Evidence

## Authority
- Architecture source: operations/blocks/B-XXX.md, section ...
- Implementation package: operations/packages/UP-XXX.md
- In case of contradiction, the confirmed block governs; do not silently reinterpret
anything.

## Boundary
- writable repos/paths:
- read-only sources:
- prohibited areas:
- no global installations / do not output secrets.

## Required Result
- complete ticket sequence;
- commits per provable increment;
- regression;
- isolated artifact run;
- evidence outputs;
- status updates in tickets/package.

## Non-Goals
- ...

## Semantic Stop Conditions
- new public responsibility;
- alternative with a different owner/source of truth;
- compatibility requirement without an evidenced consumer;
- change to a confirmed invariant.
```

```
## Working Method
1. Check boundary and baseline.
2. Create a visible plan.
3. Implement and commit tickets sequentially.
4. Capture tests and output unchanged.
5. Build the artifact run afresh and in isolation.
6. Completion audit against all acceptance criteria.

## Final Report
Commits, tests, artifact path, open points, unchanged sources.
```

The goal-prompt should enable autonomous work without optimizing away semantic stops. “Decide everything yourself” is legitimate only for editorial and technical degrees of freedom.

10.9 Template 7 — Evidence/Claims Ledger

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

10.9.1 Status Rules

- `VERIFIED` only with a directly checkable artifact.
- `INFERENCE` names the underlying evidence.
- `INTERPRETATION` is not phrased as fact.
- `OPEN` remains open even if the planned solution sounds convincing.
- `REJECTED` describes a real explored alternative.
- `REPLACED` names the old and new contracts.

10.9.2 Claims Audit

For every central statement:

1. Which exact evidence ID supports it?
2. Does the source substantiate behavior, intent, or acceptance?
3. Is the temporal snapshot appropriate?
4. Is universality being illegitimately derived from one example?
5. Is a plausible counterposition missing?

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 Requirements

- The run destination does not exist beforehand; no silent overwriting.
- The code commit/dependency version is pinned.
- Inputs are complete or reference immutable artifacts.
- every command names its input and output;
- the runtime smoke test checks more than just the build;
- the run does not change source repositories;

- sensitive values are redacted;
- repetition creates a new run or provably identical output.

10.10.2 Artifact README

```
# B-XXX — Run 001

Code revision: <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 When the Builder Itself Fails

An existing run folder is not retrospectively rewritten into a successful run. Instead:

```
run-001/
├── FAILED.md          cause, exit code, affected outputs
├── raw/               existing raw outputs
└── ...

run-002/               entirely new evidence
```

`FAILED.md` must clearly state whether the product claim was falsified or only the evidence generation was invalid. A broken heredoc, a wrong checkout, or missing Git provenance are not a product regression — but they prohibit using the run as evidence.

10.11 Template 8b — Independent Boundary/Rollback Review

```
# RV-XXX — Review <Slice>

## Authority and Source
- Target block / package:
- exact commit range:
- existing artifact run:

## Review Questions
- Does each ticket commit carry only its claimed contract?
- Which mutations can succeed on the server side yet appear
  to have failed on the client side?
- Is cleanup complete, idempotent, and transparent about errors?
- Does the run really substantiate its source revision?
- Are secrets, WIP, and the staging area excluded?

## Findings
### F-01 — <Title>
Severity:
Evidence:
Claim impact:
```

```
Required correction:

## Reproducible Review Run
- new focused tests;
- full regression;
- exact source materialization;
- raw output and checksums;
- negative results remain visible.

## Result
PASS | FIXED | OPEN | INVALID_EVIDENCE
```

The review must have a new line of attack. Merely repeating the feature builder is regression, not an independent review.

10.12 Template 9 — Manual Acceptance

```
# Acceptance — B-XXX / Run 001

Reviewer:
Time:
Artifact commit/hash:

## Preconditions
- fresh run;
- no hidden local configs;
- expected tool versions.

## Steps and Observations
1. ...
   - expected:
   - observed:
   - Status: PASS | FAIL | QUESTION

## Semantic Questions
- Does the right layer own the behavior?
- Is anything surprisingly implicit?
- Does the cause of failure remain visible?
- Does a second real input/consumer work?
- Has a non-goal accidentally been brought in?

## Result
ACCEPTED | SHARPEN | REPLACE | OPEN

## Rationale
...
```

Acceptance must not merely tick off expected outputs. At least one question must put pressure on the architecture claim itself.

10.13 Template 10 — Replacement Decision

```
# Replacement R-XXX — <old contract> → <new contract>

## Old Actual Contract
APIs, states, hooks, behavior, commit range.

## Why It Was Plausible
Strongest semantic and technical reasons.

## Contradicting Evidence
Artifact Run, Observation, User Acceptance.

## Semantic Break
```

```
Which responsibility / source of truth / authority was wrong?

## New Contract
Precise replacement rules.

## Remove
- APIs
- Flags
- States
- Fixtures
- Document assumptions

## Preserve
- Mechanics
- Tests as regression for other invariants
- Generator/Inspection
- Migration knowledge

## Compatibility Decision
None | Migration | Versioned legacy path until date X

## New Pressure Test
...
```

10.14 Template 11 — Cycle Closure

```
# Cycle Close — B-XXX

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

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

## Invariants That Have Stabilized
- ...

## Removed Assumptions
- ...

## Uncertainty Still Open
- ...

## New Pressure / Follow-Up Block
- ...

## Public Claim Boundary
What may be claimed externally, and what may not?
```

10.15 Template 12 — Delivery / Evidence / Productization

For small concrete assignments, this split may live in a single file:

```
# Delivery

## Concrete Result
What must work for this assignment?

## Acceptance
Who checks which real path?

## Non-Goals
Which platform/reuse questions do not block delivery?
```

```
# Evidence

## Recurring Pressure
What friction could be relevant beyond the assignment?

## Evidence
Where exactly did it occur? Do not present speculation about the future as recurrence.

## Next Cheap Test
What second consumer or slice could falsify the commonality?

## Status
OBSERVED | REPEATED | STRATEGIC | REJECTED

# Productization Decision

Status: NOT_DECIDED | EXPLORE | BUILD | REJECT
Owner:
Budget/Time frame:
Consumer:
Public Contract:
Support/Version boundary:
Migration:
```

`NOT_DECIDED` is a legitimate result. The evidence is not lost as a result; but the delivery assignment does not acquire a hidden platform obligation either.

10.16 Using Real Conversations as Evidence

Transcripts can show *when* a problem was recognized and *why* a direction was plausible. They are not technical authority. For a source-backed casebook, the following applies:

Class	Permitted presentation
VERBATIM	exact short wording; shortening visible; event referenced
PARAPHRASE	substantiated condensation without quotation marks
RECONSTRUCTION	several substantiated passages; do not claim a perfect real-time sequence
INTERPRETATION	explicitly mark present-day interpretation
COMPOSITE	openly identify a teaching example drawn from several cases
FICTIONAL	only as a pseudo-example, never as a historical scene
OPEN	omit attractive but not reliably substantiated recollections

10.16.1 Minimal Quote Audit

```
Quote ID
→ Session/Event ID
→ unchanged raw snapshot + SHA-256
→ exact fragment
→ manuscript location
→ privacy/context check
```

Visible assistant responses can be a source. Hidden model reasoning must not be retold. Terminal text in a user message is treated as visible transcript evidence, but technical claims drawn from it are checked against the actual test/code snapshot.

10.17 Cooperative Semantic Review in Five Moves

1. **The owner formulates the pressure**, without a desired solution yet.
2. **The partner builds the strongest counterposition**, including real benefits and costs.
3. **The owner steelmans it back:** What is right about it before disagreeing.
4. **Both shift the level:** Is the dispute about mechanics, ownership, policy, authority, or delivery?
5. **The artifact decides only its own claim:** Tests and runs are not inflated into simulated user acceptance.

A review is not better because it ends without conflict. It is better when the remaining contract is narrower and its falsification condition clearer.

10.18 Definition of Ready for an Implementation Package

A package is **ready** when:

- the initial pressure is substantiated;
- disputed terms are defined;
- owner, source of truth, and trust boundary are clarified;
- replacement versus extension is explicit;
- at least one serious counterposition has been evaluated;
- the public contract delta is named;
- non-goals bound the slice;
- acceptance and falsification conditions exist;
- the ticket slice can be committed with tests green;
- no open semantic question has merely been shifted into the goal-prompt.

10.19 Definition of Technically Implemented

- all tickets in scope committed;
- regression green;
- public contract documented;
- isolated artifact run built;
- evidence outputs present;
- required boundary/rollback review performed or explicitly justified as unnecessary;
- no uncommitted source dependency;
- no claimed manual acceptance.

10.20 Definition of Accepted

- technically implemented satisfied;
- review findings fixed, accepted, or classified as an open boundary;
- real review artifact checked manually;
- acceptance observations documented;
- countercheck/second input, where relevant;
- result `ACCEPTED` or explicit `SHARPEN/REPLACE`;
- claims ledger updated;
- remaining gaps and next pressure named.

10.21 Warning Signs of Architecture Drift

- Implementation adds a new term that does not appear in the block.
- The goal-prompt grows longer because the contract itself remains unclear.
- Tickets name files instead of observable changes.
- Tests know only the happy paths of the concrete demo.
- an adapter begins to own state.
- a reference client becomes a prerequisite for other clients.
- new flags preserve two contradictory fundamental semantics.
- “later” contains precisely the trust/concurrency boundary that would have to support the current claim.
- WIP code is mixed into the documentation as though already committed.
- Acceptance is derived from a CI result.
- a ticket commit already implements the responsibility of the next ticket;
- cleanup starts only after the mutating call has returned successfully;
- a failed artifact run is silently repaired or overwritten;
- the evidence builder checks itself only against its own assumptions of success.

10.22 Warning Signs of Premature Universality

- abstract name without a second real provider;
- shared fields come entirely from the first tool;
- unsupported is modeled as an empty value;
- target differences end up as a collection of options;
- the interface cannot uphold its error guarantee for all backends;

- “flexible” primarily means templates, includes, and overrides;
- the personal UX merely becomes configurable, but is not decoupled;
- compatibility sits in the core instead of in a separate artifact.

10.23 Exploration or an Implementable Target Design?

10.23.1 Still Exploration

- several alternatives differ in ownership;
- source of truth is open;
- the same term means different things;
- a falsification condition is missing;
- the first real E2E cannot be described;
- non-goals would obscure the central risk boundary;
- the proposed API needs hidden implicit logic.

10.23.2 Implementable

- a contract is chosen and justified;
- alternatives are documented, not merely forgotten;
- inputs, outputs, and errors can be described;
- trust/authority is clear;
- the slice crosses relevant layers;
- tickets can be committed individually with evidence;
- an artifact run can put real pressure on the premise.

10.24 Public/Company Redaction Check

- no tokens, secrets, credentials;
- no unfiltered local absolute paths;
- no personal names without purpose/consent;
- no customer, employee, or company data;
- no raw chats as a convenient source dump;
- commit and test claims match the public snapshot;
- do not present internal controls as publicly implemented;
- open acceptances remain open;
- private mappings/pseudonym tables remain separate;
- license and third-party notices checked.

10.25 30-Minute Cycle Review

For small teams, a short fixed sequence is often enough:

1. **5 min — Pressure:** What concrete finding starts the cycle?
2. **7 min — Semantics:** Owner, truth, authority, failure.
3. **5 min — Counterposition:** strongest alternative and falsification.
4. **5 min — Slice:** smallest real vertical test.
5. **3 min — Non-goals:** what is deliberately not being built?
6. **3 min — Evidence:** what outputs/acceptance do we need?
7. **2 min — Decision:** implement or explore further?

The timebox is no substitute for depth. It only prevents simple decisions from being inflated by artifact maintenance.

11 Conclusion: Architectural Knowledge Emerges Through Contradiction

The mounts worked. The lifecycle worked. The tests worked. That is precisely why these cases are strong. ROBA shows a mechanism that genuinely worked early on while its overarching meaning could have kept growing for years. DIX shows a complete contract that broke semantically precisely because of its clean artifact run. In both cases, the simple story would be wrong: Neither was everything before it nonsense, nor was the later boundary obvious from the start.

The useful story is this:

- bwrap, capability reduction, and dynamic mount propagation remained real technical evidence;
- host path identity lost to inspectable runtime truth;
- personal profiles lost their claim to universal ontology;
- Robac remained a possible reference UX, but not a generic semantics owner;
- ROBA Core became a standalone state/authority service;
- DIX began with a slice close to the UI without remaining a UI core forever;
- Elements and Datamodels became narrow capabilities;
- Compositions gained local dependency graphs and explicit functions;
- Applications made recursive composition possible;
- a redundant interface artifact type was rejected;
- the universal domain-level lifecycle was removed, its graph mechanics preserved;
- CLI became a normal module and an organizational pressure test rather than the next core feature;
- automatic contracts, Norn, Strands, and spec-authoritative bindings were actually built and partly accepted — and were nevertheless removed from the mandatory core again under a later generator counterexample;
- on the reduced core, a manually composed multi-app CLI and local modeled state held up;
- DIX integrated ROBA optionally through its public API and built an ephemeral cross-process capability registry from it;
- despite green features, an independent review found a ticket boundary violation, an ambiguous rollback path, and errors in its own evidence process.

11.1 What the Cooperation Really Contributed

It would be convenient to describe Matthias as the source of semantics and Anam as a fast implementation machine. The transcripts do not support this division of labor. Matthias designed technical models, let others convince him, and revoked his own directions. I found ownership problems, formulated hard counterpositions, built target designs — and repeatedly turned good objections into a new clean system too quickly.

The productive difference was not who was right more often. It lay in the fact that a correction did not have to be a social defeat. “Wrong track” could be accepted. “Bullshit” needed a technical justification, received one, and did not end the cooperation. On the lifecycle, I explicitly disagreed; Matthias adopted my examples and challenged their ownership conclusion. The new contract was stronger because the counterposition had been strong.

This way of working demands more thinking from the human in particular, not less. AI can develop a plausible architecture in depth within minutes. This raises the cost of poorly established meaning: It becomes consistent, larger, and harder to question more quickly. The human contribution is not the ritual final “Approve,” but taking responsibility for establishing and reopening the semantics. The machine contribution is not merely code production, but structuring, steelmanning, implementation, and checkable evidence — as long as it does not disguise its limits as certainty.

11.2 The Five Rules That Survive the Snapshot

1. **Semantics before code, but not instead of code.** An open question of responsibility must become visible; then it needs a small real slice.
2. **Evidence has multiple levels.** Regression, artifact runs, and human acceptance check different statements.

3. **Negative evidence remains a result.** A working but wrong contract is valuable if its break remains traceable.
4. **Replacement changes validity, not history.** Premature compatibility must not preserve false meaning; usable mechanics remain inventoried.
5. **Mechanism, policy, and native semantics remain distinguishable.** Composition connects them deliberately; the core does not normalize them out of convenience.

11.3 The Strongest Counterposition

ACDD can tip into ritual: long blocks, perfect ledgers, many small commits, and little product. No further template protects against that. Only asking what knowledge each artifact generates does.

If a team already develops its architecture on an evidence-bound basis through clear contracts, good experiments, formal methods, and strong review practice, it does not need the name ACDD. If a block carries no real uncertainty, it should stay short. If a ticket has no evidentiary statement of its own, it should merge with another. If an artifact run puts no architectural claim under pressure, it is build theater.

Likewise, not every small assignment may fund a platform. Delivery takes precedence over unresolved productization. But paying for recurring friction only locally and never preserving it as evidence wastes architectural knowledge. The three-way split `DELIVERY / EVIDENCE / PRODUCTIZATION` is not a new platform. It is a boundary against both extremes.

11.4 The Book as Its Own ACDD Cycle

V0 was built, text-selectable, visually checked, and reproducible. Its technical claims remain bounded by the same snapshot. It was not "wrong." The new pressure was this: The text explained the method but barely let the reader experience why we needed it at all. V1 therefore did not silently change the source revision. It added a private, hashed transcript snapshot, event and quote ledgers, an explicit classification of the montage, and the real workshop voice. The unchanged V0 PDF remains alongside this edition as a comparison artifact.

V1.1 repeats the same step at the technical level. The state from September 5 remains preserved byte for byte as V1. The new operations snapshot from September 10 is added with separate commits, tests, claims, and its own private source capture area. The earlier claim "DIX×ROBA is not implemented" was correct for V1. It is not secretly rephrased, but replaced in this edition by a new, substantiated state.

The book thus also follows the rule of knowledge we formulated earlier outside DIX: Preserving history and preserving current validity are not the same thing. The dry account remains visible. Its form does not become law.

11.5 The Next Real Cycles

The first DIX×ROBA seam now exists technically. The next technical pressure no longer lies in mere API compatibility, but at three harder boundaries:

1. human acceptance of B-017, UP-018, UP-019, and RV-001;
2. a native bwrap E2E that mounts only a minimally privileged SocketPrincipal into the child instead of owner/control tokens;
3. a security/failure review for socket reachability, revocation, unknown mutation outcomes, and cleanup under process termination.

In parallel, it remains to be checked whether local modeled DIX state holds up under a second real consumer or merely serves the current config composition well. More features would be a weaker test for this than a genuinely different owner.

For ACDD itself, the more important cycle is an independent one: Another team applies the minimal contract to an unfamiliar problem and documents not only whether code was produced, but which architecture decision was confirmed or replaced by which evidence. Only then can stronger claims be made about cycle costs, organizational impact, and transferability.

Until then, the central thesis remains a strong but limited interpretation:

> **Architecture becomes more robust when we treat it as an explicit, falsifiable, and evidence-bound sequence of decisions — and when real contradictions have the right to replace tested code semantically.**

That is less convenient than a finished method. It is more honest. And after everything that worked in these workshops and still was not yet the answer, honesty here is not a tone. It is the architecture.

12 Appendix A — Evidence and Source Register

12.1 Snapshot

The first case study claims refer to the V1 capture from **September 5, 2026, 19:21:54 UTC**. V1.1 adds a second technical capture from **September 10, 2026, 20:40:48 UTC**. The new state does not overwrite the old one. Absolute local source paths are not needed in the PDF; the following abbreviations refer to the fixed internal research snapshots.

The workshop reconstruction also uses a private Codex transcript snapshot through **20:51:16 UTC** on the date of the V1 capture. Transcripts provide evidence of the language and thought processes at the time, not independent evidence of technical system properties.

Abbreviation	Commit / Form	Content
DIX-CODE	Git `35aa9e8f`	DIX code through DX-029
DIX-OPS-C	Git `f12103d5`	committed Operations through UP-003
DIX-OPS-WIP	copy pinned by SHA-256	B-004–B-006, UP-004–UP-006, tickets/runs
ROBA-CORE	Git `bdca5fbb`	M4 Public Core Contract
ROBA-HUB	Git `a793d4b6`	Hub through Manager Introspection
ROBA-COCKPIT	Git `fba2bef4`	reference client snapshot
ROBA-TOP-C	Git `bd08b7e3`	selected concepts
ROBA-HISTORY-WIP	copy pinned by SHA-256	selected history notes
TX-ROBA	Codex rollout pinned by SHA-256	ROBA/current thread with fork lineage
TX-DIX	Codex rollout pinned by SHA-256	DIX Operations and implementation thread
TX-EVO	Codex rollout pinned by SHA-256	insight/status exploration
DIX-CODE-V1.1	Git `9f14223a`	DIX through the RV-001 rollback fix
DIX-OPS-C-V1.1	Git `15768a88`	committed Operations including the corrected RV-001 builder run
DIX-OPS-U-V1.1	copy pinned by SHA-256	all untracked Operations artifacts visible at capture
ROBA-CORE-V1.1	Git `bdca5fbb`	unchanged M4 Public Core Contract

The full transcript SHA-256 hashes, session IDs, event ranges, and selection rationales are in `research/transcript-ledger.md`. The raw files are private and are not part of the repository.

12.2 Test Evidence

System	Snapshot	Result
DIX	DX-029	209 passed, 1 deprecation warning, 3.52 s
ROBA Core	RC-016	59 passed, 32 subtests, 1 warning, 34.49 s
ROBA Hub	RC-021	14 passed, 17.71 s
V1 Quote Audit	26 entries / 34 fragments	`TRANSCRIPT_QUOTES_OK`
DIX V1.1	RV-001 fix `9f14223`	148 passed, 36.21 s
ROBA V1.1	`bdca5fb`	59 passed, 1 warning; 32 subtests, 27.19 s total

The times are observations from the local snapshot run, not performance benchmarks.

12.3 Evidence Ledger

12.3.1 DIX Foundation Through the Lifecycle Finding

ID	Status	Brief Claim	Source
E-001	VERIFIED	DIX Foundation was a browser-capable vertical slice.	B-001; DX-001–DX-004
E-002	VERIFIED	Renderer/Interactions were separated semantically.	B-002; UP-002
E-003	REPLACED	The UI/form Core was replaced by the capability Core.	B-003; DX-005–DX-008
E-004	VERIFIED	Element and Datamodel remain UI-neutral.	B-003; Core code
E-005	VERIFIED	Compositions are trusted in-process code under Module Roots.	B-004; DIX README/ code
E-006	VERIFIED	Instances have local dependency scopes and explicit Function APIs.	B-004; DX-009–DX-016
E-007	VERIFIED	Local wrappers form ownership/repair boundaries.	B-004; generator/runtime
E-008	REJECTED	An additional general Interface artifact type was rejected.	B-005
E-009	VERIFIED	Applications form owner-scoped recursive graphs.	B-005; DX-018–DX-025
E-010	VERIFIED	DIX is not a central daemon.	B-005; DIX README
E-011	VERIFIED	B-005 implemented/ tested universal hooks and graph teardown.	DX-022; DX-025; Run-001
E-012	VERIFIED	Run-001 contains a `lifecycle_only` app without functions.	B-005 Run-001 inputs
E-013	INTERPRETATION	The fixture shows structure and activation being mixed.	B-006 + E-011/E-012

12.3.2 Replacement, ROBA, and Integration

ID	Status	Brief Claim	Source
E-014	REPLACED	Hooks were replaced by structural Create/Destroy semantics.	B-006; DX-026/DX-027
E-015	VERIFIED	Typer CLI is being created as a normal first-party module.	B-006; DX-028/DX-029
E-016	VERIFIED	Datamodel, CLI binding, and function signature are separate contracts.	B-006; Typer code
E-017	OPEN	DX-030 and full B-006 acceptance were open at the snapshot.	HEAD/ticket/block status
E-018	VERIFIED	DIX DX-029 ran with 209 passing tests.	raw pytest output
E-019	VERIFIED	ROBA M4 Core contains no Actions/Events/Execution/Workspace policy.	README; RC-016
E-020	VERIFIED	ROBA separates Daemon, Control, Context, and Capability sockets.	README; RC-005–RC-012
E-021	VERIFIED	TokenPrincipals and SocketPrincipals have parallel Resource ACLs.	README; RC-008/RC-012
E-022	VERIFIED	Bootstrap was removed from Core; external composition is evidenced.	RC-014–RC-016
E-023	VERIFIED	Hub manages recovery as an external consumer.	Hub README; RC-017–RC-021
E-024	INTERPRETATION	ROBA's reduction contradicts universality derived from personal UX.	history + current contract
E-025	VERIFIED	DIX integrates ROBA unidirectionally through its Public API.	UP-018/UP-019; module code
E-026	REPLACED	The V1 status "integration not implemented" was replaced by the V1.1 slice.	both snapshot boundaries

12.3.3 Transcript and Narrative Evidence

ID	Status	Brief Claim	Source
E-027	VERIFIED	The live-mount E2E and immediate mechanics boundary are evidenced in the visible history.	TX-ROBA; S-01; TQ-001–002
E-028	VERIFIED	Both sides visibly correct and disagree at several architecture boundaries.	S-02, S-04, S-06, S-12
E-029	VERIFIED	DIX began with a small form-oriented task, not a universal platform assignment.	TX-DIX; S-15
E-030	VERIFIED	The Operations block was established as a living derivation artifact before tickets/tests.	S-09; TQ-017
E-031	VERIFIED	B-005→B-006 contains a strong lifecycle counterposition and the subsequent ownership pivot.	S-12; TQ-020–023
E-032	INTERPRETATION	ROBA experience may have encouraged DIX's semantic shifts.	scene/chronology comparison; no measurement of causality
E-033	VERIFIED	Published chat fragments have been automatically checked against the private index.	quote manifest and verifier

12.3.4 The Second DIX Arc and the Real DIX×ROBA Seam

12.3.4.1 Building the Contract

ID	Status	Brief Claim	Source
E-034	VERIFIED	B-007 completed Python signature, FunctionContract, one-shot runner, and Auto-Typer with 231 tests.	B-007; UP-007; Run B-007; `831d5ed`
E-035	VERIFIED	B-008 introduced Norn/Strand/Knot, ran with 263 tests, and received human acceptance.	B-008; UP-008; Run B-008; `7e8bb6c`
E-036	VERIFIED	B-009 technically implemented declarative contracts as authoritative bindings; 89 tests passed.	B-009; UP-009; Run B-009; `e300d47`
E-037	VERIFIED	UP-012 connected Datamodel endpoints to Norn bindings and technically ran with 108 passing tests.	B-012; UP-012; `4bde81c`

12.3.4.2 Replacement and Rebuilding

ID	Status	Brief Claim	Source
E-038	REPLACED	Generator evidence refuted Datamodel compatibility as a global admission gate; the mandatory Norn/binding/contract chain was removed.	B-013; UP-013; DX-056–DX-058
E-039	VERIFIED	The reduced B-013 Core calls raw Python functions and ran with 78 passing tests after replacement.	UP-013; `de00b83`
E-040	VERIFIED	B-014 built a manually composed multi-application Typer CLI on the reduced Core; 100 tests, Core unchanged.	B-014; UP-014; `2a07c6a`
E-041	VERIFIED	B-015 documents Operations drift, establishes the Router/AGENTS cutover, and prohibits retrospective backfill.	B-015; Operations-AGENTS/README
E-042	VERIFIED	B-016/UP-017 deliver composition-local modeled state with `get`/Full`set`; 132 tests; human acceptance completed.	B-016; UP-017; Run UP-017; `46f3e0c`

12.3.4.3 Integration and Review

ID	Status	Brief Claim	Source
E-043	VERIFIED	UP-018 integrates ROBA optionally through its Public API, local config, Application, and Typer; 141 DIX and 59/32 ROBA tests.	B-017; UP-018; Run UP-018; `b03b447`
E-044	VERIFIED	UP-019 builds `dix.control`, root/manager sockets, and cross-process rebind; 147 DIX tests.	B-017; UP-019; Run UP-019; `f11a60b`
E-045	VERIFIED	RV-001 found functionality implemented ahead of its ticket and an ambiguous socket-create rollback path despite passing features.	RV-001; DX-072/DX-073; `9f14223`
E-046	VERIFIED	RV-001 Run 001 was retained as invalid; Run 002 evidenced 5 review, 148 DIX, and 59/32 ROBA tests. A follow-up commit removed the staging leak; Run 003 evidenced the corrected builder with ten successful exit artifacts and no residual staging.	RV-001 Run 001–003; Operations `37fd868`, `a868a34`, `15768a8`

12.3.4.4 Snapshot and Claim Boundary

ID	Status	Brief Claim	Source
E-047	VERIFIED	An independent V1.1 rerun from fresh Git checkouts yielded 148 DIX and 59/32 ROBA tests; an archive-only attempt was invalid because Git provenance was missing.	private V1.1 source snapshot; raw test outputs
E-048	OPEN	At capture, B-017, UP-018, UP-019, and RV-001 have no documented final human acceptance.	respective status/acceptance sections
E-049	VERIFIED	V1.1 pins DIX `9f14223`, Operations `15768a8`, ROBA `bdca5fb`, and, separately, all visible untracked Operations files.	snapshot metadata and SHA-256 manifest
E-050	INTERPRETATION	The sequence 263 → 89 → 108 → 78 tests is not a decline in maturity, but shows changing contracts and the removal of false mandatory semantics.	E-035–E-040; Claims C-018

12.4 DIX Commit Register

12.4.1 Foundation and Composition

Commit	Ticket	Key Statement
2080a9b	DX-001	Config/CLI Foundation
f9437c7	DX-002	Models, Registry, Components
bf7a247	DX-003	Server, Renderer, Runtime State
08b6709	DX-004	E2E Demo and 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 and Inspection
75e1e51	DX-011	Module Registry
a10e769	DX-012	isolated Composition instances
0609921	DX-013	startup lifecycle at the time
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 and the First Lifecycle Boundary

Commit	Ticket	Key Statement
9e9607f	DX-018	Composition lifecycle replacement during Application expansion
ec0726a	DX-019	Application Spec/Inspection
356ac58	DX-020	Atomic Mixed Module Loading
79ac3b5	DX-021	isolated Application instances
5a39afa	DX-022	transactional Application lifecycle
32b8f4f	DX-023	Application Scaffolding
3508b62	DX-024	Application Generator
bb9c5cf	DX-025	recursive Application pressure test
39bb762	DX-026	lifecycle-neutral Core
b073220	DX-027	Destroy before Module Unload
9a5549c	DX-028	strict Boolean type
35aa9e8	DX-029	declarative Typer Composition

12.4.3 Second Arc — Selected Target Commits

Commit	Ticket	Key Statement
`831d5ed`	DX-036	automatic CLI Application pressure test
`7e8bb6c`	DX-041	Strand-driven Typer pressure test
`e300d47`	DX-046	static contract Application pressure test
`4bde81c`	DX-055	model-bound Norn execution
`de00b83`	DX-058	contract-free seed launchers after Core replacement
`2a07c6a`	DX-061	composed multi-application CLI
`46f3e0c`	DX-067	Local State boundary pressure test
`b03b447`	DX-071	DIX×ROBA Source/Wheel/Runtime E2E
`f11a60b`	DX-075	Cross-Process Capability Rebind
`9f14223`	RV-001	rollback and boundary hardening

12.5 Source Index by Chapter

Book Chapter	Primary Sources
Architecture as a Process of Inquiry	B-003, B-004, B-005, B-006; Claims Ledger
Executable Cycle	DIX Operations README; B-006 flow section; UP/ticket structure
DIX Case Study	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; commit chain; selected concepts/thread
DIX Second Arc	B-007–B-017; UP-007–UP-019; RV-001; DX-030–DX-075
DIX × ROBA	B-017; UP-018/UP-019; RV-001; module code and cross-process E2E
Workshop Scenes / Cooperation	Transcript Ledger; Narrative Evidence Ledger; quote manifest
Company / Small Assignment	S-13, S-15; B-001/B-006; Claims C-014

13 Appendix B — Glossary and Term Matrix

13.1 Glossary

Acceptance
Evaluation of a real artifact by an authorized human role. Not equivalent to a passing test run.

ACDD
Architecture Cycle-Driven Development. In this book: the development of architecture through explicit, bounded, evidence-bound cycles.

Application (DIX)
A recursively composable, trusted Python artifact made up of Applications and Compositions with an explicit local Function API. In the snapshot, it has no universal domain-level Core lifecycle.

Architecture Block
The living working truth of an open architecture question. Holds pressure, alternatives, history, target design, invariants, and open boundaries.

Artifact Run
An isolated, reproducible run with fixed inputs, commands, outputs, and a review path that puts a complete architecture claim under real pressure.

Authority
Permission to legitimately execute an operation or state change. Not identical to identity or transport access.

Capability
A concretely usable ability with a named access and error boundary. In ROBA, a reachable Capability socket can also count as deliberately authorized access.

Claims Ledger
A list of central statements with epistemic status, evidence, and boundaries.

Compatibility Artifact
A separate module that places a deliberately smaller shared contract over native target systems. It is not automatically Core.

Component (DIX Core)
A narrow internal capability such as `element`, `datamodel`, `composition`, `application`, or `module`.

Composition (DIX)
A trusted Python code building block made up of Components and other Compositions, described by a spec and `runtime.py:Runtime`, with a local dependency scope and an explicit Function API.

Construction / Destruction
Structural construction or teardown of a definition/instance graph. Not automatically domain-level starting, stopping, or cleanup.

Context (ROBA)
A logically isolated state and ACL boundary within a named daemon.

Control Authority (ROBA)
Permission to manage the Context Registry through `control.sock`. Not identical to a Context Owner token.

Core
The layer that owns mechanics required by all consumers and under its control. “Small” is a consequence of clean ownership, not an end in itself.

DIX
In the snapshot: Declarative Interface eXecutor; a system for narrow Core capabilities, trusted Compositions, and recursive Applications.

Delivery / Evidence / Productization Three deliberately separate tracks: delivery of a concrete assignment, preservation of potentially cross-cutting architectural insight, and an explicitly funded reuse or product decision only after that.

Evidence Chain

A traceable connection between pressure, decision, implementation package, tickets, commits, tests, artifact run, independent review, and acceptance.

Function Contract

An explicitly exposed ability with a signature, error boundary, and ownership boundary. In the current local DIX runtime path, real Python methods remain authoritative for their signatures; specs own graph assembly and function exposure. Other boundaries may deliberately establish an IDL as the authority.

Goal-Prompt

An executable work assignment that translates confirmed architecture into autonomous implementation. Not an independent source of truth.

Instance (ROBA)

A named state scope within a Context. No nested Context/process semantics.

Capability Registry (DIX × ROBA)

An ephemeral management artifact owned by DIX in the ROBA Context `dix.control`. It holds root and target Context credentials and projects bounded tokenless rebind through SocketPrincipals. No new ROBA Core type and no durable secret persistence.

Independent Review / RV

A separate review assignment after feature evidence, posing new questions about ticket boundaries, rollback, source provenance, and the evidence builder. It does not always have to be carried out by different people, but must not merely repeat the feature run.

Invariant

A deliberately established boundary to be protected across multiple changes.

Locator (ROBA)

A typed address `id:<id>` or `unix:<absolute-path>` for Daemon, Control, or Context; scope uses `id:<scope>`.

Module (DIX)

A trusted delivery and dependency bundle under a configured Module Root. May contain Compositions and Applications; has no domain-level lifecycle.

Local Modeled State (DIX)

An optional Composition capability made up of a declaratively resolved Pydantic model and exactly one local model instance. Exposes only complete `get()` and validating full-model `set()`; no global state registry, persistence, or event semantics.

Norn / Strand / Knot (historical, replaced)

B-008 terms for a composition-scoped contract registry and execution (`Norn`), a named input/output contract (`Strand`), and an implementing or composing Composition (`Knot`). The approach was actually implemented and accepted, but was replaced as mandatory Core execution in B-013.

Native Capability

Target-system-specific, honest functionality that does not hide differences under a premature shared interface.

Non-Goal

An explicit boundary of a cycle. Must not be used to hide a difficulty that is crucial to the claim.

Owner-Scope (DIX)

The assignment under which a local instance graph is constructed and destroyed.

Principal (ROBA)

A token- or socket-based access principal with Resource ACLs.

Pressure Test

A targeted real slice that puts an architecture hypothesis under pressure across relevant integration boundaries.

Reconstruction An editorial assembly of multiple evidenced transcript passages and artifacts. It may condense, but must not claim an uninterrupted real-time sequence or invented dialogue.

REJECTED
An explored alternative was deliberately rejected.
REPLACED
A real earlier contract was replaced by another. History and usable evidence remain visible.
ROBA
In the M4 snapshot, a small local ephemeral Context State service with named daemons, Multi-Context, State, Instances, and Resource ACLs.
SocketPrincipal (ROBA)
A Unix Capability socket with the same ACL structure as a TokenPrincipal. The socket access is the credential; no additional token is used.
Technically Implemented
Status for committed scope with regression, an artifact run, and technical evidence. It is explicitly not equivalent to human `ACCEPTED`.
Source of Truth
The authoritative source of a state or contract. Snapshots and projections are not automatically second truths.
Transcript Quote (TQ) A short `VERBATIM` fragment from a visible user/assistant message, verifiable through event ID, text hash, rollout line, and private snapshot. It provides evidence of wording, not automatically of technical truth.
Structural Lifecycle
Definition/load, graph construction, destruction, and unload of the references owned by Core.
Domain-Level Lifecycle
Activation, readiness, retry, pause, drain, stop, or cleanup of a concrete system. Belongs in Core only if its semantics are universal and controllable by Core.
Trusted Module Root (DIX)
A configured path under which Python modules are admitted for in-process execution. No substitute for a sandbox.
Implementation Package
A binding translation of a stabilized architecture block into scope, non-goals, ticket boundaries, and acceptance.
Vertical Slice
A bounded but complete chain from real input through contract and execution to observable output and teardown.

13.2 Term Matrix

Term	Owns Structure?	Owns Behavior?	Owns Authority?	Typical Owner
DIX Module	yes	no, delivers code	Trust Admission	DIX Module Component
DIX Composition	yes	explicit functions	via Host/Dependency	local runtime graph
DIX Application	recursive	explicit functions	via Host/Backend	Owner-Scope
DIX Interface (historical)	exposure	bound calls	Session/Host	outer adapter; no longer Core
DIX Local State	model instance	`get` / Full-`set`	local graph	Composition instance
DIX ROBA Module	config + adapter graph	Daemon/Registry functions	ROBA principals	DIX Application/Composition
ROBA Daemon	processes/listeners	state service	OS + Control	daemon process
ROBA Context	State/Instances	primitive state ops	ACL	Context Owner
ROBA Hub	recovery metadata	management flow	Manager Owner + Control	external consumer
bwrap Module (planned)	Namespace/Mounts	process start	OS Capability	Linux module
Delivery Artifact	concrete assignment	precisely required workflow	assignment/acceptance	delivery team
Evidence Candidate	observed pressure	no shared API yet	research/architecture process	named observer
Productized Capability	versioned contract	maintained functions	product/capability policy	separate owner

14 Appendix C — Known Gaps and Next Stages of Development

14.1 Evidence Gaps

1. **Historical V1 state:** B-004 and B-006 had open manual acceptance reviews on September 5. V1.1 does not rewrite that historical status boundary.
2. **DIX second arc:** B-013 and B-014 are technically implemented, but their overall domain-level blocks had not received final acceptance at the time of the new snapshot.
3. **DIX × ROBA:** UP-018 and UP-019 are technically implemented and supported by source, wheel, and cross-process runs. B-017, both packages, and RV-001 await final human acceptance.
4. **ROBA history:** Capsules are selective reconstruction, not a complete project chronicle.
5. **ACDD externally:** No independent comparative or organizational study.
6. **Transcript reconstruction:** Three rollouts and eight lineage sessions used cover the selected scenes, not the complete project history. The assignment of inherited events to sessions is deterministically inferred from the fork start and UUIDv7 time.
7. **Effect of cooperation:** The visible scenes show productive counterpositions, but are not a controlled study of human–AI teams.
8. **Delivery/Productization:** The three-way division has a substantive derivation, but its costs have not yet been evaluated across multiple company projects.
9. **V1.1 operations capture:** Several historical operations artifacts were still untracked. They are fully and separately hashed, but their missing Git status remains part of the evidence boundary.

14.2 Technical Topics Outside V1.1

- Remote/network transport for ROBA;
- Windows-native ROBA or isolation transports;
- security audit of DIX trusted module loading;
- package distribution, versioning, and migration;
- production observations on performance and long-term stability;
- native DIX-bwrap-E2E with a minimal ROBA-SocketPrincipal;
- concrete DIX modules for Docker or Windows;
- persistent or encrypted recovery of the ephemeral DIX-ROBA registry;
- comprehensive security/failure audit of capability socket distribution;
- UI for operations/evidence navigation;
- automated consistency checking between block, ticket, commit, and claim.

14.3 Methodological Development Stages

14.3.1 V1.2 — External Reader Critique

- Test terminology without project knowledge;
- add counterpositions from experienced architects;
- use templates on an outside slice;
- clarify the unclear distinction between block and ADR.

14.3.2 V1.3 — Second Independent Case Study

An independent architecture cycle outside DIX/ROBA with the same evidence chain. The goal is to test transferability, not to confirm it.

14.3.3 V1.4 — Company Overlay

- RACI;
- security/compliance mapping;
- signature and approval rules;
- cost measurement;
- AI data classification;
- public redaction pipeline.

14.3.4 V2 — After Real-World Application

Only after at least one independent, complete cycle outside the DIX/ROBA workshop:

- Sharpen terminology;
- compare with established RFC/ADR/architecture test approaches;
- anti-patterns from actual misuse;
- measurable cycle and review costs;
- an updated DIX/ROBA case study only with a new, explicit snapshot.

Do not mistake this for a gap: ACDD is not intended to become a universal execution framework, a ticket tool, or a prompt library. Tooling can support the flow, but must not own the semantics. The most important open work is real-world application — not more meta-infrastructure.