Predicting Before Building: A Pre-Registration Protocol for Architecture Research, and What One Instance Corrected

Yoni Lavi
Codeliance

August 2026

Note on process. This paper was developed collaboratively with Claude (Anthropic), which served as primary drafter under the author's direction, and the artifact it accounts for was likewise AI-implemented against author-directed specifications. The architectural vision and synthesis are the author's; the literature survey, formal framing, prose, and implementation were produced by the AI and verified against primary sources and a test suite. This accounting is itself part of the paper's subject: a protocol for reporting what was predicted before it was built has to be explicit about who did the predicting and who did the building.

Abstract. Software architecture research has a credibility problem that is structural rather than cultural: designs are almost always described after they are built, so a reader cannot tell which claims were predictions and which were rationalisations. Adjacent fields answer this with pre-registration. This paper reports a protocol that adapts it to architecture work — freeze the design vision before implementation, publish it under a citable identifier, guard it against silent edit by a check that runs in the build, permit only dated errata, and then report outcomes prediction by prediction without revising the predictions — and reports one complete instance of running it. The instance is a signal-graph architecture whose founding vision was published and frozen before a demonstrator existed; the demonstrator is reported separately [1], and this paper accounts for it against nineteen predictions, of which seven are substantiated at the demonstrator's scale, five hold only under restrictions the vision did not attach to them, three remain exactly as conditional as first stated, and four were not attempted. The protocol's value showed up where we did not expect it: not in confirming predictions but in the four design corrections the artifact forced, each invisible from the vision's vantage point and each recoverable only because the original text could not be quietly amended. Revocation turned out to be ill-posed until capability *identity* was expressible; trust checking and edge typing turned out to be one order rather than two; "no ambient authority" turned out to name two different guarantees of which only one is worth claiming; and the cheapest convention for a composition boundary turned out to be the one hardest to verify. We are explicit that this is a single instance, that the protocol's mechanisation is cheap while its discipline is not, and that a freeze guard prevents silent revision without making anyone honest.

1 Introduction

A software architecture paper almost always describes a system that already exists. The design is presented as a set of choices, the choices are justified, and the justifications are persuasive — because they were written by people who already knew which choices had worked. A reader has no way to separate the predictions from the rationalisations, and the author frequently has no way either: the memory of what one expected before building is not reliable, and nothing in the ordinary publication process preserves it.

This is not a complaint about honesty. It is a structural property of writing up work after doing it, and adjacent fields have converged on the same structural answer. Clinical trials pre-register endpoints; psychology and, increasingly, empirical software engineering use registered reports, in which a protocol is reviewed and accepted before results exist. The mechanism is always the same: fix the claim in a form that cannot later be edited, then report against it.

Architecture research has been slow to adopt this, and the reason is worth naming, because it is a real obstacle rather than mere inertia. A registered report fixes a **hypothesis and an analysis plan**. An architecture vision is neither: it is a design, a set of properties the design is expected to have, and a set of hedges about which of those properties are actually established. It cannot be reduced to a statistical pre-specification. What it can be reduced to is a **document**, frozen at a point in time, whose claims can be enumerated afterwards and whose text can be prevented from changing.

This paper reports a protocol built on exactly that reduction, and one complete instance of running it. The protocol is described in Section 2 and its mechanisation — the part we think is the contribution — in Section 2.1. The instance is a research programme on capability-injected signal graphs: a founding vision [2] published and frozen before any of it was built, and a demonstrator, reported separately [1], built afterwards to test it. Section 3 gives the prediction-by-prediction accounting, Section 4 reports the four design corrections the artifact forced, and Section 5 sets out what remains. Section 6 states the limits, of which the first and largest is that one instance is one instance.

1.1 What this paper claims

> Pre-registration is adaptable to architecture research at low cost: a design vision can be frozen, published under a citable identifier, and mechanically guarded against silent revision by a check that runs in the ordinary build, after which outcomes can be reported claim by claim without the predictions moving. Doing so changes what the write-up can contain — the corrections a design needs become reportable results rather than invisible edits — and this paper offers one worked instance as evidence that the cost is small and the yield is not.

What it does not claim: that the protocol has been validated. One instance, run by the author on the author's own research, is not evidence that the protocol generalises, improves accuracy, or survives contact with a team that disagrees internally. Nor does the mechanism make anyone honest — a freeze guard prevents silent revision; it does not prevent a vague prediction, a self-serving reading of an outcome, or a corpus of predictions chosen because they were likely to hold. Section 6 returns to each of these.

2 The protocol

The protocol has five steps, and each exists to close a specific way the reporting could otherwise drift.

1. **Freeze the vision before building.** The design document is finished, dated, and fixed at a commit before any implementation exists. What makes this meaningful is the "before": the founding vision here was frozen at the last repository state that contained no executable demonstrator, so its claims could not have been informed by one.
2. **Publish it under a citable identifier.** The frozen document is posted to an archival venue and takes a DOI. This matters more than it appears: a frozen file in a repository is only as

durable as the repository, whereas a published record is a third-party copy the author cannot alter. In the instance reported here, the vision is a Zenodo preprint under CC BY 4.0.

3. **Guard the freeze mechanically.** A check runs in the ordinary build and fails it if the frozen sources drift from the freeze commit. This is the step that converts an intention into a property, and Section 2.1 is about it.
4. **Permit dated errata, and nothing else.** A frozen paper acquires corrections through an append-only errata record rather than in-place rewriting. A single, narrowly-bounded editorial revision for publication is allowed — typography, phrasing, metadata — on the explicit condition that it changes no prediction, hedge, or argument, and it too is recorded.
5. **Report outcomes without revising predictions.** The accounting paper states, claim by claim, what the artifact did to each. Predictions that failed stay in the frozen text as they were written; the accounting says they failed.

The fifth step is the one with teeth, and its rationale is worth stating plainly. Predictions edited after the outcomes are known are unfalsifiable. The interesting content of an implementation — which conditional claims survived contact with an artifact, which needed weakening, which proved harder than the hedge admitted — is legible only when the predictions stand as first made. A vision paper quietly updated to match what got built reads better and is worth less.

2.1 The mechanisation, which is the cheap part

Everything above could be a convention. Conventions decay, and they decay silently, which is the failure mode a reader cannot detect. The protocol's mechanisation is three small pieces of ordinary build tooling, and its cost is measured in hours.

The freeze guard. A script compares the frozen paper's sources against their content at the recorded freeze commit and exits non-zero on any difference. It runs as a pre-commit hook and in the build, so an edit to a frozen paper fails before it can be committed, not during review. The guard's pointer is updated only by the deliberate, recorded act of re-freezing at a publication commit.

Self-containment of frozen inputs. A frozen paper that renders figures from a live shared artifact is not actually frozen: its text is fixed while its evidence moves underneath it. The frozen vision here therefore carries its own pinned copies of every graph definition and diagram source it references and builds only from them, so later evolution of the shared artifact cannot alter what the frozen document shows. This is the subtlest of the three and the one we got wrong first.

One owner per number. The accounting paper interpolates no measurements. Every figure it refers to is cited to the demonstrator paper, whose own figures are interpolated from a single reproducible run of an evaluation harness. Two documents that can each state a number are two documents that can disagree; the corpus permits exactly one source for any given measurement.

None of this is technically interesting, which is the point. The barrier to pre-registering an architecture design is not tooling difficulty — it is that nothing forces the discipline, and a check in the build forces it at negligible cost.

2.2 Why a design vision, and not a hypothesis

The unit of pre-registration here is a **document**, and the claims are extracted from it afterwards. That is weaker than a registered report, where the claims are enumerated in advance and the analysis plan is fixed, and the weakness is real: enumerating claims after the outcomes

are known leaves room to choose a flattering granularity, to split a failed prediction into a substantiated part and a not-attempted part, or to quietly omit one.

Two things bound that room, neither of which eliminates it. The frozen text is published, so a reader can check the enumeration against the source. And the accounting uses four statuses of which two are unflattering by construction — a claim is **conditional** if it is exactly as unproven as when written, and **not attempted** if no evidence exists in either direction — so the cost of an honest "we did not do this" is low enough that there is little incentive to disguise it. A protocol that made every non-result embarrassing would produce fewer of them, and would be worse.

3 The instance: predictions and outcomes

The founding vision [2] argued that AI coding agents have removed the historical obstacle to graph-based code representations — the human preference for text — and that this opens a development model built around a *signal graph*: a functional reactive program in which each node is a function from typed inputs to typed outputs, all authority is held as explicitly declared capability handles, and trust is a propagating type-level annotation. In that model the graph is simultaneously the architecture model, the security policy, and the program. The paper was written before any of it existed and hedged accordingly: its security claims were stated as properties that *would* hold in a sound realisation of a type system that did not yet exist.

The demonstrator built to test it is reported in full separately [1]. In outline it comprises canonical graph definitions, a dependency-free validator implementing six classes of graph-level analysis, an executable runtime that instantiates nodes with injected capability handles, and two enforcement tiers: host-level object discipline, and a confined tier compiling node bodies to WebAssembly components whose imported interfaces are derived from their declared capabilities. This section states what it does to each prediction.

Four statuses are used. **Substantiated** means the artifact backs the claim at the demonstrator's scale, with a test or a measurement behind it. **Partial** means the claim holds under a stated restriction the vision did not attach to it. **Conditional** means the claim is unchanged: still hedged, still unproven, still stated in the mood the vision used. **Not attempted** means no work was done and no evidence is offered either way.

Prediction from the founding vision	Status	Evidence in [1]
Unsafe wirings are rejected statically, before execution	Substantiated	mutation corpus
Trust cannot be laundered by relabelling the consumer	Substantiated	mutation corpus
A node cannot exceed its declared capabilities	Partial	tier comparison
Nodes have no ambient authority	Partial	tier comparison
Per-crossing overhead stays inside the stated envelope	Substantiated	overhead benchmark
Prompt injection is structurally attenuated	Substantiated	injection scenario
Hierarchical composition follows from the model	Substantiated	sub-graph execution
Capability identity, revocation, and rotation are expressible	Substantiated	identity and caretakers
Implementations are interchangeable across languages	Partial	confined tier
The *declared* capability distribution is complete and reviewable	Substantiated	validator
User-level authorisation threaded through capability injection	Partial	principal binding
Behavioural contracts constrain generated implementations	Partial	contract layer
Well-typed wiring implies noninterference	Conditional	—
The full type system (graded or decentralised labels)	Conditional	—
Memory-level unforgeability on CHERI hardware	Conditional	—
Replay and time-travel debugging from boundary logs	Not attempted	trace, but see Section 3.3
Differential evaluation of the graph at scale	Not attempted	—
Agent tooling for the graph workflow	Not attempted	—
The visual graph editor	Not attempted	see Section 3.3

Table 1: The founding vision's nineteen claims and what the demonstrator does to each. Every "Substantiated" is qualified by the demonstrator's scale — one graph, one domain, a curated corpus — as Section 6 and the demonstrator paper's own limitations section both state.

3.1 What the demonstrator substantiates

The load-bearing prediction — that a graph carrying capability and trust annotations admits a static analysis strong enough to reject unsafe wirings before anything runs, implementable with modest tooling rather than a research type system first — holds: a dependency-free validator rejects the corpus mutations at assembly time, each by the analysis meant to catch it. Its sharper form, that trust cannot be laundered by relabelling the consumer, also holds, and is the result we would keep if forced to keep one: a graph in which **every edge type-checks** is still rejected, because trust is an order applying uniformly to edges and node bodies rather than a property of edges. The vision predicted this would need a flow-sensitive discipline rather than

falling out of edge typing; it was right, and the demonstrator shows the cost of getting it wrong is invisible at the level of edge types.

The performance envelope held with room to spare, against an assertion the vision made with no numbers behind it at all: the overhead benchmark [1] reports puts a crossing well inside it while marshalling typed values rather than opaque bytes. Prompt-injection attenuation held in exactly the form claimed — attenuation, not elimination — and the residual the vision warned about is the one that was measured. Hierarchical composition is where the artifact was more generous than the prediction: "falls out naturally from the model" reads as optimism, but proved accurate in a checkable way, because a sub-graph cannot provision authority of its own — not because a rule forbids it, but because the executor holds no backend to provision from, so the confinement we expected to have to enforce had no mechanism by which it could fail.

3.2 What it substantiates only in part

Five claims hold under restrictions the vision did not attach to them, and stating the restrictions is more useful than restating the claims.

A node cannot exceed its declared capabilities is true on the confined tier and false on the host tier, where a hostile node can read the filesystem, open a socket, or read the environment. The vision described capability injection as though enforcement followed from it. It does not. Enforcement follows from the **artifact**, and only nodes compiled to components get it; the host tier's escapes are recorded as passing tests precisely so this does not read as universal.

Nodes have no ambient authority holds on the confined tier in a stronger sense than the vision articulated, and the discrepancy is the subject of Section 4.

Implementations are interchangeable across languages is demonstrated for several node bodies spanning the typed capability boundary, each running from a Rust body regenerated from the same signature and contract as its Python counterpart with the graph unchanged. It stays partial, and the restriction is the point: a contract is a partial specification, so two implementations can satisfy it and still differ observably, and what the demonstrator holds its bodies to is a vocabulary too small to pin their behaviour. Existence across several nodes and capability kinds is not a claim about the general case.

Behavioural contracts constrain generated implementations was recorded as conditional until the demonstrator acquired a contract layer, and it is worth being exact about how far that moves it, because the distance between **checking** and **verifying** is the whole restriction. Nodes carry pre- and postconditions over a closed predicate vocabulary; a failure assigns blame, indicting the wiring for a broken precondition and the body for a broken postcondition; and a specification the vocabulary cannot express is refused before anything runs rather than quietly evaluating to false. What none of that does is establish a property over all inputs: contracts are checked on the values a run happens to produce. The vision's word was "verified", and the artifact checks. The closed vocabulary is a second restriction of the same kind — a postcondition relating output length to input length is inexpressible, and widening the language far enough to state it would need a solver.

User-level authorisation threaded through capability injection has the same shape: bound, not typed. A run may bind an authenticated principal at assembly; a node declared `binds_principal` — mirroring `discharges_trust` exactly — is the sole point licensed to extend the chain of parties acting on that principal's behalf; the chain is recorded per node and at every altitude in the delegation shape of RFC 8693; and a sub-graph acts for whoever its parent

acts for, because it is assembled with the parent's principal and has no means of minting one. Those are tests rather than sentences. The restriction is that the scope lives on the capability **instance** and in the trace, not in the capability's **type**, so the validator cannot answer without running the graph what set of principals a node could ever act as — which is the property the confused-deputy argument ultimately wants. Attenuation is not implemented at all: a node inherits the principal it was given and cannot narrow it.

3.3 What remains conditional, and what was not attempted

The type-system claims are untouched. No soundness argument connects well-typed wiring to noninterference; the two-point lattice is a first realisation, not a proof, and the choice between two-point, graded, and Jif-style decentralised labels [3] is open. The vision's conditional mood for these claims was correct, and the demonstrator paper keeps it rather than upgrading it to the present tense used around it — which is the discipline most easily lost when everything adjacent reports a built result, and the one the protocol is least able to enforce mechanically.

CHERI integration is untouched, so enforcement stops at the WASM boundary. The two claims that were conditional here in an earlier state of this accounting — behavioural contracts and user-level authorisation — have since moved to **partial** on the strength of a contract layer and a principal binding the demonstrator did not previously have (Section 3.2); what they have not acquired is the static half of either, which is the half the type-system claims above would supply.

Two rows deserve their own note, because both are places where something adjacent exists and the honest verdict is still **not attempted**.

Replay. The demonstrator emits a structured execution trace: per node, its enforcement tier, the trust labels of its input and output, and the distinct capability crossings it made. That is precisely the structured log the vision's replay claim assumed, and having it is a genuine prerequisite. But recording is not re-execution. A journal of what crossed which boundary is evidence that crossings *can* be journalled, not a mechanism that re-runs the graph from them, and no replay was attempted. The trace substantiates the narrower adjacent fact and does not move this verdict.

The visual editor. The repository ships a web inspector that renders the canonical graphs, triggers real runs on either tier, and overlays the returned traces. It is the reviewing half of the predicted tooling, built against the real runtime rather than a mock — and it does not author: no node is added, no edge rewired, no `with` clause granted from the interface. The vision predicted a visual **editor** inside an agent workflow, in which an agent proposes a graph transformation and a human reviews it visually. Since the editing side is untouched and the agent workflow entirely so, the verdict here is **not attempted**, with the inspector noted rather than credited. An earlier draft of this accounting recorded it as **partial**; that was the demo flattering the prediction, and the correction is itself an instance of the discipline this paper is about.

4 What building it revealed that the vision did not see

This is the section that justifies the protocol, and it is worth being explicit about why. Seven substantiated predictions are a pleasant result and a weak one — the author chose the predictions and the author built the artifact, so agreement is cheap. What is not cheap is a correction: a place where the artifact contradicted the design and the design had to move. Those are the results that a frozen prediction makes reportable, because without the frozen

text the correction would simply have been an edit, and nobody would ever have known the earlier position existed.

There are four.

Revocation had an unnoticed prerequisite. The vision listed capability revocation as an open problem and reached for the standard answer, caretaker indirection [4] — without noticing that revoking a **specific** handle first requires a way to **name** one, since the natural default shares one handle per capability **type** across every node declaring it. Capability **identity** had to become expressible in the graph source before revocation was even a well-posed operation. The vision listed the consequence and missed the prerequisite, which is a characteristic failure of designing by analogy to a known solution: the analogy imports the mechanism and drops its preconditions.

Trust was one rule, not two. The vision described edge type-compatibility and a trust-discharge check as separate obligations. Implementing them showed they are the same no-upward-coercion order applied in two places, and that treating them separately is exactly what admits the laundering graph — the one in which every edge type-checks and the fault is invisible at the level of edge types. This simplified the design rather than complicating it: a case where the artifact argued the vision into a cleaner position, and one that a post-hoc write-up would have presented as the intended design all along.

Confinement is a property of an artifact or of a configuration, and the vision conflated them. An early build of the confined tier compiled nodes to WASI modules run under an empty context — no preopens, sockets, environment, or clock. That confined them, but the modules still **imported** `environ_get`, `fd_write`, and `path_open`: the imports were present and merely powerless, so confinement was a fact about how the host had configured the runtime, and a misconfigured host would have silently granted them. Rebuilt with no WASI adapter, the components import no such function at all. Both states satisfy the sentence "nodes have no ambient authority" and only one survives a misconfigured deployment. This is the clearest instance we have of a security claim that sounds identical in prose while naming two different guarantees, and only building it twice distinguished them — which is an argument for implementations, not for pre-registration; what pre-registration adds is that the vision's undifferentiated phrasing is still on the record, so the distinction is legible as a correction rather than as something we always knew.

The union-alias convention for sub-graph outputs needed a check the vision did not call for. The vision treated the union convention as the simplest of several designs for collapsing a sub-graph's terminal types into a single boundary type, and moved on. In the demonstrator it proved the cheapest to adopt and, at first, the least self-checking: the boundary type was asserted in the graph source and verified against nothing, so a sub-graph node could misdescribe what it emits and no analysis would object. Building it surfaced that the property most worth checking — that a boundary type honestly describes the terminals behind it — was exactly the one the convention dropped, and that the fix was cheap once seen. The lesson generalises past this instance: a boundary type that abbreviates must still be verified against what it abbreviates.

Three of these four are corrections to **security** claims, and all four share a shape: the vision was wrong not about whether a property could hold but about what the property **was**. That is a failure mode prose is particularly bad at exposing, because a sentence like "nodes have no ambient authority" reads as one claim and is two.

5 The forward agenda

The demonstrator establishes that the graph-level analyses and capability confinement are implementable. It does not deliver the language, the proofs, or the tooling. This section sets out what remains, in three phases of increasing scope with a realistic dependency ordering: Phase 1 designs the language and its type system, Phase 2 hardens the result for meaningful deployment, and Phase 3 addresses the formal and hardware questions. Where the demonstrator has narrowed an item, the narrowing is stated so the remaining question is not overclaimed as smaller than it is.

5.1 Phase 1: the language and its type system

The signal graph language. Design the capability-annotated signal graph language: its type system, its expression of trust tainting, its composition rules. The target is a language expressive enough to encode realistic system architectures while remaining amenable to visual rendering and agent manipulation. Arrowized FRP [5] and algebraic effect systems [6] are the primary formal references. A key decision is the degree of dependent typing required: Idris 2 or Agda for full expressiveness, or a more restricted system (a Haskell-like type system with phantom types for trust levels) for tractability. The demonstrator uses the restricted system; the Phase 3 verification work may require the full one.

The trust lattice, and the conditions it must satisfy. The demonstrator realises a two-point lattice `Untrusted ⊑ Trusted` with no upward coercion and discharge as the sole sanctioned upward move. What remains open is the choice among two-point, graded (`Untrusted ⊑ Sanitised ⊑ Trusted`), and Jif-style decentralised-label [3] designs. The condition any successor must meet can be stated concretely, and is exactly what a locally-sound-but-non-compositional label system loses: the flow relation used by the wiring check (`required ⊑ provided`, with a node's output label a monotone function of the meet of its input labels) must be **transitive**, so that a path of individually well-typed edges is itself well-typed, and **monotone**, so that composing nodes cannot manufacture trust the parts did not have. Compositionality of noninterference follows from standard results in information-flow security [7, §5], but the signal graph's wiring model must be shown to satisfy the conditions those results require — an adaptation, not a mechanical application. The two-point lattice satisfies both trivially, which is precisely why it is weak evidence for a richer one.

Error handling, conditional flow, and fan-out. The demonstrated graphs show basic conditional routing and error routing, but real systems require richer patterns: fan-out to multiple consumers, error propagation chains, and fallback logic. Arrowized FRP provides combinators for choice and fan-out (`ArrowChoice`, `&&&`), but their integration with capability annotations and trust tainting has not been worked out. Fan-in is where the trust lattice's monotonicity condition first becomes load-bearing rather than trivial, since a merge must combine labels rather than propagate one. A graph language that cannot express "on payment failure, notify the user and log the error" without escaping to imperative code would not be viable.

Node-local state. Many real components need persistent local state between invocations: session caches, rate-limit counters, accumulated aggregations. In FRP, state is modelled through feedback loops and signal accumulators; the interaction between stateful combinators, capability annotations, and the deterministic replay property has not been analysed, and a node maintaining implicit internal state may violate the assumptions that enable the replay and verification claims. The language must define whether state is an explicit feedback edge, a

stateful combinator, or a capability-mediated external store — and which choice preserves the security and replay properties.

Hierarchical capability routing. When a parent provisions a capability to a sub-graph, the sub-graph must route it to the internal nodes that require it, and only those. Three designs are visible: (i) a flat parameter list the parent matches by type, with internal fan-out by convention; (ii) named capability slots the parent binds explicitly; (iii) structural matching on capability types, routing handles automatically to every matching `with` clause. Option (i) is settled **at the runtime**, which is the narrowest available claim: it is implementable and sufficient to run the composition, not demonstrably the right choice. It is the option requiring no new language surface, which is why a demonstrator can reach it without prejudging the design — and a flat list matched by type is also exactly where aliasing ambiguity lives, which is the argument for (ii) or (iii). The demonstrator's per-node identity map resolves the **naming** step without committing to any of the three, and carries identity across a single composition level rather than an arbitrary hierarchy.

Sub-graph output aggregation. The dual problem at the output side is more open in its **design** than the input side, though the demonstrator has closed the checking half. When a sub-graph's terminals emit distinct types, the boundary must collapse them into the single type the parent declares. The options mirror the input side: (i) define the boundary output as the union of terminal types, so no aggregation node is required; (ii) named output ports the parent wires individually; (iii) an explicit aggregation node inside the sub-graph. The demonstrator runs on (i) with the output side now checked against the referenced graph's terminals, so the convention is verified against what it abbreviates rather than merely asserted. What stays open is the design — whether (ii) or (iii) is preferable — together with the multi-terminal case, where a run reaches several terminals and there is no single boundary value, which the runtime refuses rather than resolves.

User-level authorisation, in the type system rather than at the instance. The confused-deputy attenuation argument depends on user-scoped capabilities being bound at the graph boundary and propagated downstream as part of the injected handle. The demonstrator does that dynamically (Section 3.2): a principal is bound at assembly, a declared node may extend the delegation chain, the chain crosses composition boundaries, and all of it is recorded in the trace. Three things remain. The scope belongs in the capability's **type**, so that the set of principals a node could act as is a question the validator answers rather than a fact a run reveals. Attenuation — a node narrowing the principal it was handed — is unimplemented, and the authorisation literature is clear about its price: systems that keep delegation analysable, from Cedar's decidable fragment [8] through Biscuit's append-only Datalog [9] to the policy intersection of production IAM, restrict narrowing to a monotone, enumerable operation, while those admitting unrestricted dynamism make the authority topology unknowable until run time. And the interaction between a typed principal dimension and the trust lattice is unexamined, though the trade is the same one already made for trust, which is a reason to expect the design to transfer and no evidence that it does.

Graph evolution and type-system evolution. When a node's signature changes, downstream consumers may break; the language must define compatibility rules for signature evolution and support versioned interfaces at sub-graph boundaries to enable independent team ownership. Distinctly, the type system will itself need to evolve — new trust levels, new capability kinds, refined subtyping — and migration of existing graphs across type-system versions, with preservation of verified properties, is an open problem the design should anticipate.

Agent tooling and developer experience. Build the workflow that takes a natural-language change description, proposes a graph transformation, generates node implementations, and submits them for automated verification, extending existing SDD tooling to operate on typed graph artifacts rather than prose. Alongside it, the visual graph editor and diff viewer: capability edge additions, trust boundary crossings, and sanitisation gaps must be visually salient. The demonstrator's inspector narrows the **viewing** half — rendering, running, and trace overlay exist against the real runtime — so the open editor-side question is specifically **authoring**: graph edits made from the interface with the validator in the loop, and the diff view over them. It narrows nothing about the agent workflow, and nothing about **graph-scale comprehension**, which the demonstrator does nothing to address — its concrete example is a single small graph and real systems have hundreds of nodes, a gap [1] states with the empirical figures behind it. Hierarchical decomposition is the expected approach, but the interaction between hierarchical abstraction, capability wiring, and trust propagation across sub-graph boundaries has not been designed, and a type error deep in a sub-graph's wiring must produce an error message comprehensible at the level the developer is working at.

5.2 Phase 2: hardening and deployment

Shallow verification. Develop tooling to confirm that generated node implementations satisfy their declared contracts. The demonstrator has the contracts (Section 3.2) and none of the verification: predicates are evaluated on the values a run produces, so nothing is established over all inputs, and the generative mode that would derive a property-based-test strategy from a precondition and fuzz a body against it is designed and unbuilt. Three techniques cover complementary concerns. *Property-based testing* establishes type-level invariants across random inputs; *contract testing* checks node-local pre- and post-conditions on representative traces, giving concrete evidence for properties the static type system cannot express; and *architectural fitness functions* [10] verify cross-cutting properties of the assembled graph — capability-scope constraints, trust-zone integrity, absence of forbidden wirings — that belong to the composition rather than any single node. The obligation is deliberately bounded: type conformance, contract satisfaction, and graph-level structural invariants, not arbitrary program-property verification.

Contract incompleteness. Bounding that work is a problem the demonstrator cannot reach but which limits the code-as-compiled-artifact claim directly. Contracts are partial specifications: two implementations can satisfy the same contract and still differ observably — in latency, in resource consumption, in which variant they emit on inputs the contract does not pin down. Regeneration can therefore change production behaviour within the slack the contract leaves, with no verification step failing. Tightening contracts narrows the slack at the cost of authoring burden and verification tractability. Characterising which classes of behavioural difference are operationally significant, and what contract discipline makes regeneration safe with respect to them, is open.

Generated implementations, tested as such. Every node body in the demonstrator was written by hand, so the model's central workflow claim — that implementations are generated against typed signatures and are interchangeable artifacts — is untested by the artifact that would make it checkable. The machinery is now in place: a node's permitted import set is derived from its `with` clause and compared against its built component, so a generated body reaching for authority it was not granted fails before shipping. How often that happens, and whether the guard is the operative constraint on agent-authored code, is the obvious next experiment and the one we expect to be most informative.

Event log infrastructure and replay fidelity. Design the structured event logging that capability boundary crossings produce automatically, define the formal conditions under which replay fidelity holds, and characterise the classes of failure that violate it. The assumption that a boundary event log is a complete and deterministic record holds for single-threaded deterministic nodes and degrades for nodes with internal concurrency or timing dependencies. Fan-in is the inter-node instance of the same problem: where several upstream nodes feed one consumer, arrival order is a scheduling artifact rather than a topological property, so the log must record the realised merge order and the runtime must reproduce it. Perfect replay is not the claim; materially better fidelity than conventional logging, with the failure classes characterised and managed, is.

Migration path. Design the incremental adoption route for existing systems. The minimal entry point is wrapping an existing service as an opaque node with a declared capability signature — a boundary describing what the service **does** without requiring internal restructuring, analogous to declaring a foreign function interface. Over time an opaque node can be decomposed into sub-nodes with narrower signatures. The demonstrator's two-tier composition is evidence that such intermediate states are workable, and it also shows why naming the graduation honestly matters: opaque wrappers and host-tier nodes provide **architectural visibility** — their authority is explicit and reviewable — while enforcement applies only to nodes actually ported to the confined tier. The benefit curve is graduated: visibility at the first step, structural enforcement earned as wrappers are decomposed and confined.

Covert channels. A node granted a permitted capability can encode information into its legitimate outputs — the choice of query, the timing of invocations, the shape of emitted events — through channels the type system does not model. The `Untrusted<T>` discipline addresses explicit data flow, and noninterference results handle the class of flows the type system observes; general covert-channel elimination is a known-hard problem and is not a target. The pragmatic aim is characterisation: naming which channels the model closes, which it narrows, and which remain out of scope.

Revocation and rotation, completed. The demonstrator implements targeted, opt-in revocation and rotation on the host tier and carries revocation across to the confined tier. Two parts remain: sandbox-tier **rotation** — re-pointing over the same typed boundary — and the graph-transformation or redeployment form of both operations, which is undesigned. This is operations-adjacent work that the migration story needs.

5.3 Phase 3: formal foundations and hardware

Type-system soundness. The security properties depend entirely on the type system being sound: every well-typed graph must satisfy noninterference and capability confinement. This is the obligation the demonstrator most conspicuously does not discharge, and a soundness bug would propagate to every layer of the defence-in-depth stack, since the runtime's capability injection is configured by the type system's analysis. Before a mechanised proof for the core calculus, property-based testing of the type system itself — random graph generation with expected type errors, fuzzing of the wiring checker — is the cheap validation. A first slice of this runs today: the demonstrator's test suite generates random trust labellings over the two-point lattice and asserts that an edge is rejected on trust grounds exactly when it is an upward coercion. That is a worked instance of the discipline, not a substitute for it.

Compilation correctness. For the compilation from signal graph semantics to capability-restricted WASM, confirm that component boundaries, capability signatures, and trust

annotations are preserved across the production boundary. This is a bounded correctness claim about a well-defined transformation, closer in kind to CompCert [11] than to general program verification — but CompCert took a decade. The realistic near-term target is a mechanised preservation proof for a simplified subset of the language, sufficient to validate the approach and identify the hard cases. The minimal set of invariants required to guarantee the security properties in the production runtime must be identified before the full scope is fixed.

CHERI integration. Design the mapping from architectural capabilities — typed handles injected at node boundaries — to CHERI hardware capabilities at the memory level, using CHERI's fine-grained compartmentalisation to enforce node isolation below the WASM boundary, and characterise graceful degradation on non-CHERI hardware. CHERIoT [12] provides a reference architecture; the WASM Component Model's capability interfaces provide the natural software interface above which CHERI enforcement is applied. This is what would move the confined tier's enforcement from unforgeable-at-the-WASM-boundary to unforgeable-at-the-memory-level.

Distributed authority. The signal graph controls capability flow within a deployment, but a node wired to a network capability can communicate with any reachable service and potentially acquire authority out-of-band that the graph does not model. The E language [4] and Agoric's Hardened JavaScript [13] address this through reference-based communication discipline; this work inherits the same open question for the distributed case. Scoping to a single deployment boundary is the pragmatic approach for Phases 1 and 2; the distributed extension is a later research question.

6 Threats to validity

6.1 One instance, and whose

This is a single case study, run by the author on the author's own research. Nothing here shows that the protocol generalises, improves predictive accuracy, or is workable for a team whose members disagree about what was predicted. A protocol evaluated by the person who designed it, on work they also authored, is the weakest evidential arrangement available, and the four corrections of Section 4 are reported by the same party who would have been embarrassed by concealing them.

The predictions were chosen by the predictor. Nineteen claims were enumerated from the frozen text after the outcomes were known. A different enumeration would produce different counts, and the granularity at which a claim is split materially changes how flattering the table looks — a prediction that fails wholesale can be split into a substantiated part and a not-attempted part by anyone willing to do so. The mitigation is only that the frozen text is published, so the enumeration is checkable; the incentive is not removed.

A freeze guard prevents silent revision, not self-serving reading. The mechanism enforces that the prediction's text is unchanged. It enforces nothing about whether "substantiated" was applied generously, whether a hedge is being read as weaker than it was written, or whether an inconvenient claim was omitted from the enumeration entirely. Section 3.3 records one instance where an earlier draft of this accounting **did** read generously — recording the visual editor as partial on the strength of a viewing-only inspector — which is evidence that the failure mode is live rather than hypothetical, and that catching it depended on a reviewer rather than on the tooling.

6.2 Threats from the method

The demonstrator was implemented by an AI agent against author-directed specifications, and both papers were AI-drafted under the same direction (see the note on process). Two consequences deserve naming. An artifact and its evaluation produced by the same process risk agreeing with each other for reasons unrelated to the truth of the claim — which is why the demonstrator's evaluation harness pins verdicts **and reason classes** and rejects any divergence, rather than reporting whatever it finds. And a paper drafted alongside the artifact it describes is exposed to drift between prose and code, which is why the demonstrator's figures are interpolated from a single run rather than transcribed, and why the properties it asserts are, wherever practical, backed by tests rather than by sentences.

There is a third consequence specific to this paper. An AI drafter has no memory of having predicted anything, so the frozen document is doing more work here than it would for a human author reconstructing their own past beliefs — the text is the only record, which strengthens the case for the protocol while making this instance less representative of the usual case it is meant to serve.

Neither measure substitutes for independent replication, which has not occurred.

6.3 What the protocol costs

The mechanisation is cheap and the discipline is not. Freezing a document early means publishing claims that later look naive, in a form that cannot be tidied; the four corrections of Section 4 are, read uncharitably, four things the vision got wrong, and a protocol whose main output is a public list of one's own errors has an adoption problem that no amount of tooling addresses. We think the trade is worth making and note that we have made it exactly once.

7 Conclusion

The protocol reported here is not sophisticated: freeze the design before building it, publish it where it cannot be altered, guard the freeze in the build, allow only dated errata, and report outcomes without revising predictions. Its mechanisation is a few hundred lines of build tooling. What it changes is not the rigour of the research but what the write-up is **able to contain** — with the original text fixed and public, a correction becomes a reportable result instead of an invisible edit.

That turned out to be where the yield was. The seven substantiated predictions in Table 1 are the least interesting rows in it: the author chose the predictions and directed the build, so agreement between them is cheap. The four corrections of Section 4 are not cheap, and none of them was visible from the vision's vantage point. Revocation was not a well-posed operation until capability **identity** was expressible in the graph source. Trust checking and edge typing were one order rather than two, which simplified the design. "No ambient authority" named two different guarantees — one a property of the host's configuration, one a property of the artifact — and only the second is worth claiming. And the cheapest convention for a composition boundary was the one hardest to verify, because it abbreviated exactly what most needed checking. Three of the four are corrections to security claims, and all four share a shape: the vision was wrong not about whether a property could hold, but about what the property **was**.

We would not claim more than one instance supports. The protocol has been run once, by its designer, on research they also authored; the enumeration of predictions was made after the outcomes were known; and a freeze guard prevents silent revision without preventing a

generous reading — as Section 3.3 records, one row of this very accounting was initially read too generously and was corrected by review rather than by tooling. What the instance does show is that the cost is low enough not to be the obstacle. If the reason architecture designs are described only after they are built is that nothing forces otherwise, then a check that runs in the build is a surprisingly large part of the answer.

Annex A: Areas for collaboration

This work is an invitation. The synthesis it describes spans several domains that no single team is likely to cover. This annex identifies the expertise each phase requires, as a guide for potential collaborators.

Phase 1: the language and type system

- **Type theory and functional programming:** algebraic type systems, arrowized FRP, algebraic effect systems. Experience with Haskell, Idris, or Agda. The signal graph's type system is the foundation; getting it wrong here propagates to every later phase.
- **Systems programming:** WASM Component Model and WASI toolchains, capability-based I/O models, runtime implementation in Rust or C++.
- **AI agent tooling:** structured agent workflows, tool use, MCP (Model Context Protocol).
- **Developer experience design:** visual graph editors, diff viewers, reviewer cognitive load. This is as important as the formal foundations.

Phase 2: hardening

- **Formal methods:** property-based testing (QuickCheck, Hypothesis), contract testing, lightweight specification (TLA+, Alloy).
- **Distributed systems and observability:** structured logging, distributed tracing, causal ordering [14], OpenTelemetry.
- **Security engineering:** capability-based security, supply chain threat models, prompt injection as an attack class.

Phase 3: formal foundations

- **Proof assistants:** Coq or Lean 4 at theorem-proving level, for the bounded compilation correctness claim.
- **Computer architecture:** CHERI instruction set architecture (ISA) extensions, CHERIoT hardware-software co-design. The Cambridge CHERI group is the primary external knowledge source.

Bibliography

[1] Y. Lavi, "Confinement by Construction: Capability Surfaces Derived from an Architecture Model." 2026.

[2] Y. Lavi, "Architecture as Program: Capability-Injected, Model-Driven Software Development in the Age of AI Agents." [Online]. Available: https://doi.org/10.5281/zenodo.21473361

[3] A. C. Myers and B. Liskov, "A decentralized model for information flow control," *ACM SIGOPS Operating Systems Review*, vol. 31, no. 5, pp. 129–142, Dec. 1997, doi: 10.1145/269005.266669.

[4] M. S. Miller, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control," PhD Thesis, 2006. [Online]. Available: https://www.semanticsc

holar.org/paper/Robust-composition%3A-towards-a-unified-approach-to-Shapiro-Miller/f59585e6405581fe7d4cad04648d6bd8a587901a

[5] H. Nilsson, A. Courtney, and J. Peterson, "Functional reactive programming, continued," in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell (Haskell '02)*, Pittsburgh Pennsylvania: ACM, Oct. 2002, pp. 51–64. doi: 10.1145/581690.581695.

[6] D. Leijen, "Type directed compilation of row-typed algebraic effects," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, Paris France: ACM, Jan. 2017, pp. 486–499. doi: 10.1145/3009837.3009872.

[7] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003, doi: 10.1109/JSAC.2002.806121.

[8] J. W. Cutler *et al.*, "Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization," in *Proceedings of the ACM on Programming Languages*, 2024. doi: 10.1145/3649835.

[9] Biscuit contributors, "Biscuit Authorization: Specifications." [Online]. Available: https://doc.biscuitsec.org/reference/specifications.html

[10] N. Ford, R. Parsons, and P. Kua, *Building Evolutionary Architectures: Support Constant Change*. O'Reilly Media, 2017.

[11] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, doi: 10.1145/1538788.1538814.

[12] S. Amar *et al.*, "CHERIoT: Complete Memory Safety for Embedded Devices," in *56th Annual IEEE/ACM International Symposium on Microarchitecture*, Toronto ON Canada: ACM, Oct. 2023, pp. 641–653. doi: 10.1145/3613424.3614266.

[13] Agoric, "Hardened JavaScript (SES — Secure EcmaScript)." [Online]. Available: https://github.com/endojs/endo

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978, doi: 10.1145/359545.359563.