# Architecture as Program: A Capability-Injected Signal Graph and What Its Demonstrator Substantiates

Yoni Lavi
*Codeliance*

July 2026

**Note on process.** This paper was developed collaboratively with Claude (Anthropic), which served as primary drafter under the author's direction, and the demonstrator it reports 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 transparent accounting reflects the paper's own thesis: that the interesting artifact is the *intent*, not the implementation, and that honest attribution of AI contribution is preferable to ambiguity.

**Abstract.** AI coding agents have removed the thirty-year obstacle to graph-based code representations: the human preference for text. Earlier work [1] argued from that premise for a development model in which the primary artifact is a *signal graph* — a functional reactive program with explicitly typed capability boundaries — serving simultaneously as architecture model, security policy, and source of truth, and predicted that a class of vulnerability would become ill-typed rather than merely discouraged. This paper reports a demonstrator built to test that prediction, and evaluates it. The demonstrator comprises a graph validator implementing six classes of static analysis over canonical graph definitions, 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 WIT interfaces *are* their declared capabilities. We find that unsafe wirings are rejected at assembly time, each by the analysis it is meant to exercise — including trust laundering, which type-checks on every edge and is caught as a lattice violation rather than a type mismatch; that a capability-boundary crossing costs 55.6 µs, inside the 1 ms envelope the vision asserted without evidence, even though every crossing now marshals typed values; and that escape attempts succeeding on the host tier are refused on the confined tier, where an inference-only node's component imports no filesystem, socket, environment, or clock interface at all. We are equally explicit about what does not follow. The corpus is curated and illustrative, so the counts are not a soundness result; no noninterference proof is attempted; the host tier's escapes are a real gap, reported rather than omitted; and prompt injection is *attenuated, not eliminated* — adversarial text survives in a permitted free-text field, and what bounds the damage is the capability scope rather than the type. The contribution is an existence proof at modest scale that graph-level capability analysis and confinement are implementable, together with a precise account of the distance remaining to the vision's stronger claims.

## 1 Introduction

For thirty years, a small community of researchers pursued an idea that working software developers consistently rejected: that source code should be a structured graph, not text in files. Projectional editors [2], structure editors [3], model-driven architecture [4]: the history of these efforts is a history of elegant ideas that foundered on a single obstacle: programmers prefer text editors, value syntactic flexibility for half-formed thoughts, and resist representations that constrain expression before intent is clear.

That obstacle has moved. The rapid adoption of AI coding agents (Claude Code [5], Codex [6], Cursor [7], Windsurf [8]) has introduced a new primary author of implementation code that does not *require* the specific affordances of text editing that motivated human resistance to structured representations. What agents benefit from instead is semantically rich, machine-readable representations of intent that prevent entire categories of error before generation begins. Recent work on LLM code generation within the Hazel typed-hole environment [9] gives direct empirical evidence: providing agents with rich static context from the type system substantially improves generation quality. The graph representation that human developers rejected may be precisely what agent-authored software requires — and the parallel emergence of spec-driven development frameworks (OpenSpec [10], GitHub's Spec Kit [11], AWS's Kiro [12]) is evidence that practitioners have already begun treating structured intent, rather than generated implementation, as the artifact that matters.

Earlier work [1] argued that these trends converge on a development model built around a *signal graph*: a program written in a functional reactive style 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; implementation code inside nodes is a compiled artifact. That paper was written before any of it had been built, and said so: its security claims were stated conditionally throughout — properties that *would* hold in a sound realisation of a type system that did not yet exist.

This paper reports what happens when that model is built.

## 1.1 The central claim

The claim this paper defends is deliberately narrower than the vision's, and is chosen to be falsifiable by the artifact accompanying it:

> A signal graph with capability-annotated, trust-annotated types admits a practical static analysis that *rejects at assembly time* — before any node executes — a class of wirings that conventional architectures can only discourage by review, including the trust-laundering case that type-checks on every edge; and node bodies can be confined so that a node's declared capabilities are the totality of what its artifact can reach, at a capability-boundary cost of tens of microseconds.

What this claim does not say is as important as what it does. It does not say the system is provably secure: no soundness argument connects well-typed wiring to noninterference here, and that gap is Phase 3 work (Section 7.3). It does not say prompt injection is prevented: it is attenuated, and Section 4.3 reports the residual that survives on the confined tier. It does not say the corpus of caught mistakes is complete: it is curated, and counts of caught mutations are evidence of implementability, not of coverage (Section 8). A reader who finishes this paper believing a stronger claim has been shown has been misled by it, and we have tried throughout to make that outcome difficult.

## 1.2 Contributions

- **A graph validator** implementing six classes of analysis over canonical graph definitions — edge type-compatibility, a flow-sensitive trust lattice, variant completeness, capability narrowing at composition, intra-graph consistency, and cross-graph signature matching — in which trust laundering is a violation of the *same* no-upward-coercion order that governs edges, rather than a rule bolted on beside edge typing (Section 3.2).

- **An executable runtime with two enforcement tiers** that composes in one graph: host-level object discipline, and a confined tier compiling node bodies to WASM components for which a node's `with` clause *is* its component's import set, so confinement is a property of the artifact rather than of the host's configuration (Section 3.3).
- **Operational hierarchical composition**: a node whose name resolves to another graph executes by nested assembly, from which capability confinement across the composition boundary follows from the plumbing rather than from a rule (Section 3.5).
- **Capability identity, revocation, and rotation** expressed in the graph source — identity and revocation enforced on both tiers, rotation on the host tier — resolving a prerequisite the vision did not notice its own revocation problem had (Section 3.4).
- **An evaluation** — corpus verdicts pinned to the reason class that must catch them, boundary overhead against the vision's stated envelope, and prompt-injection outcomes on both tiers — emitted as a reproducible artifact so that every figure in Section 4 traces to a single run, and a regression is caught rather than silently absorbed.
- **An explicit predictions-and-outcomes accounting** against the vision's predictions (Section 5), separating what the artifact substantiates from what remains conditional.

### 1.3 Relation to the earlier paper

The earlier paper [1] set out its design and its security claims before any of it had been built, and hedged them accordingly. This paper does not revise those predictions to match what was found; it reports, prediction by prediction, what the artifact does to each (Section 5). The separation is deliberate: predictions edited after the outcomes are known are unfalsifiable, and the interesting content — which conditional claims survived contact with an implementation, which needed weakening, and which proved harder than the hedge admitted — is legible only when the predictions stand as first made.

The remainder of the paper is organised as follows. Section 2 presents the design, developed from the earlier paper and restated here to make the paper self-contained. Section 3 reports what was built. Section 4 evaluates it. Section 5 gives the predictions-and-outcomes accounting. Section 6 positions the work against prior art, Section 7 sets out the forward agenda including the problems that remain open, and Section 8 states the limits of what has been shown.

## 2 Design

This section presents the design the demonstrator implements. It develops the design set out in the earlier paper [1], restated here so that this paper stands alone; readers of that paper may skim to Section 3. Two things have changed in the restatement, and both are marked where they occur: claims the demonstrator has since substantiated are stated in the present tense, and claims it has not are left in the conditional mood they were written in. Section 5 collects that accounting in one place.

The model has four interlocking properties.

**The signal graph as source of truth.** The primary artifact that humans author, review, and version-control is a functional reactive program (FRP — see Section 2.1 for the paradigm and its history): a directed graph in which each node is a function from time-varying typed inputs to time-varying typed outputs, with any effects mediated through explicitly declared capability handles. This graph is simultaneously the architecture model, the security policy, and the program. It *is* the implementation at the level of abstraction humans author and review; the code inside each node is a generated artifact derived from it.

**Capabilities as declared requirements.** Nodes have no ambient authority: side effects (database access, network calls, LLM invocations, event emission) are not capabilities that code can reach for from the surrounding environment. They are typed capability handles declared separately from data inputs using a `with` clause — for example, a node with signature `(OrderRequest) → OrderConfirmation with DBHandle<'orders', read-write>, EventEmitter<'order-events'>` can read and write orders and emit order events, and nothing else, because no other mechanism exists within its scope. A node with no `with` clause is a pure function of its inputs in the conventional sense.

**Code as compiled artifact.** AI agents generate the implementation of each node to satisfy the behavioural contracts encoded in the graph's type signatures. The imperative code inside each node is an implementation detail, analogous to compiled bytecode. It can be regenerated, refactored, or rewritten without changing the system's meaning, provided it satisfies its contracts — an interchangeability claim that holds up to the expressiveness of those contracts, a limit examined in Section 7.2. Humans review graph transformations; they do not routinely review generated code.

**Security by construction.** Because capabilities are injected and the type system propagates trust annotations, security properties are intended to be structural invariants of the graph rather than aspirations enforced by code inspection. The strength of the resulting guarantees varies by vulnerability class and is worth naming up front. Injection attacks that depend on untrusted input reaching an interpreter in executable form would be statically ill-typed in a sound realisation. Privilege escalation — a node acquiring capabilities it was not given — would be prevented by construction, because the graph would be the complete and sole description of the capability distribution. Prompt injection and confused-deputy patterns are structurally attenuated through capability scoping and trust propagation rather than wholly eliminated; the precise strength of each guarantee, together with its schema-design prerequisites, is detailed in Section 2.6, and what the demonstrator measured of it in Section 4.3.

## 2.1 Functional reactive programming as the conceptual core

The signal graph model is not a novel metaphor. It is a direct application of *functional reactive programming* (FRP), a paradigm with a nearly thirty-year research history, here elevated from a UI programming technique to a whole-system architectural substrate.

### 2.1.1 FRP: a brief account

FRP was introduced by Elliott and Hudak's 1997 *Functional Reactive Animation* [13], which modelled interactive animations as pure functions over continuous time — *behaviours* (time-varying values) and *events* (discrete occurrences) composed without state mutation or callbacks. The denotational semantics (a behaviour is literally a function `Time → Value`) give equational reasoning unavailable in callback-driven styles. Wan and Hudak [14] gave the paradigm a rigorous stream-based semantics, and, building on Hughes's arrows [15], Nilsson, Courtney, and Peterson developed *arrowized FRP* [16], which makes *signal functions* the composable unit and their interface explicit in the type: a signal function `SF a b` transforms a signal of `a` values into a signal of `b` values. This is the formal object we extend with capability annotation.

The idea has been realised widely. Elm [17] brought strict purity and runtime-managed effects to browser UIs; RxJS [18] and Cycle.js [19] carried the stream lineage into JavaScript, the latter with a driver discipline close to capability injection — a component not given a DOM or HTTP driver simply cannot perform those effects, lacking only fine-grained capability typing and trust propagation. A parallel graph-based tradition runs through the synchronous dataflow languages

(Lustre [20], Esterel [21], Signal [22]), which compile reactive graphs to formally verifiable code for avionics and nuclear instrumentation, and flow-based programming [23], whose black-box components exchanging data across named ports anticipate the componentised form — though without static typing, capability, or trust. LabVIEW [24] and Simulink [25] validate "graph as program" at commercial scale, while illustrating the comprehension problems (Section 7) it brings.

More recently, *differential* computation — evaluating only the nodes a change affects — has been shown practical at database scale by differential dataflow [26] (Materialize [27], DBSP [28]); this work's claim that the signal graph can serve as a production substrate, not merely a development-time abstraction, depends on that line.

**A concrete illustration.** To motivate the extensions developed in Section 2.1.2, consider a node that processes user-submitted text and passes it to an LLM with tool-calling access. In a conventional system, this is dangerous: if the submitted text contains adversarial instructions, the LLM may execute them using its tools. The vulnerability is not a bug in any individual component; it is an architectural property: the unintended flow from untrusted input to a privileged executor.

In the signal graph, the same scenario is expressed as two nodes. The first, `UserInputHandler`, has signature:

```
UserInputHandler : (HTTPRequest<'POST', 'user:message'>) → Untrusted<UserMessage>
```

The second, `LLMOrchestrator`, has signature:

```
LLMOrchestrator : (SanitisedPrompt) → AgentResponse
  with LLMClient<[respond, lookup]>
```

A direct wiring from `UserInputHandler`'s output to `LLMOrchestrator`'s input is a *type error*: `Untrusted<UserMessage>` does not match `SanitisedPrompt`. The graph cannot be assembled without an explicit node that transforms `Untrusted<UserMessage>` into `SanitisedPrompt` — a node whose existence is visible in the architecture, whose implementation is subject to contract verification, and whose presence is required by the type system rather than by a policy document. In a well-typed realisation of this model, the prompt injection vulnerability would be *ill-typed*: no well-typed graph could express it. The type system design that would deliver this guarantee is the central obligation of Phase 1 (Section 7.1); the example illustrates the target property, not a proven result.


Figure 1: The direct wiring of the two nodes above is structurally rejected: the output type `Untrusted<UserMessage>` does not inhabit the input type `SanitisedPrompt`. An explicit sanitisation node — not shown — would be required by the type system to close the gap. Security, here, would be a property of graph shape.

### 2.1.2 Extending FRP: capability annotation and trust tainting

The step from FRP as a UI technique to FRP as a whole-system architectural model requires two extensions that the existing literature does not fully address.

The first is *capability annotation*. Standard FRP treats effects as values managed by the runtime, but does not give them a fine-grained type structure that distinguishes, say, a read-only database handle from a read-write one, or a sanitised string from an untrusted one. The object-capability model [29] provides the missing ingredient: capabilities are unforgeable typed references whose possession is the proof of authorisation. Combining FRP's signal graph semantics with the object-capability model's typed authority gives us signal graphs in which data-flow and capability-flow are both first-class, typed, and statically checkable.

The second is *trust tainting*. Data entering the graph from untrusted sources (user input, third-party API responses, LLM outputs) carries a type marker that propagates through signal transformations until it passes through an explicitly designated sanitisation node. This is analogous to taint tracking as studied in information-flow security [30], and specifically to the labelled-IO approach demonstrated by practical information-flow control libraries such as LIO [31], but expressed as ordinary type-level propagation rather than a separate analysis. A node that accepts `Untrusted<string>` and a node that accepts `LLMClient<[respond, lookup]>` cannot be directly wired; the type system would prevent the combination. The graph topology would enforce the security property without separate analysis.

### 2.1.3 Time as a structural dimension

In a conventional program time is implicit and history is lost unless logged. Under the purity guarantees the runtime would enforce, time would instead be structural: every signal would carry a history, and the system's behaviour at any point would be a pure function of its inputs up to that point. Two consequences would follow. Proposing a change would mean forking the graph's timeline — an agent explores the fork and, if satisfactory, merges it, discarding it otherwise at no cleanup cost because the fork is a value, not a mutation — so human review becomes a *behavioural comparison of two timelines* rather than a diff of two static models. And in production, every capability-boundary crossing (a database read, network call, or LLM invocation) would be a typed, observable event whose structured log — subject to the fidelity limits of Section 7 — would make past behaviour substantially reproducible, so debugging is replay-and-fork and the log doubles as a regression suite authored for free. None of this is built (Section 5).

## 2.2 The signal graph

The primary artifact is a version-controlled, typed signal graph with the following structure.

**Nodes** are functions with explicit signatures. A node's signature has two parts: its *data inputs* (typed signals from upstream nodes) and its *capability requirements* (typed handles to external resources, declared with a `with` clause). The data inputs describe what the node transforms; the capability requirements describe what authority it has. This separation reflects a lifecycle distinction: data signals flow at runtime as the graph propagates, while capabilities are provisioned when the graph is instantiated. A node with no `with` clause is a pure function of its inputs; a node with a `with` clause is pure with respect to its inputs *given* its handles, with all effects mediated through those handles.

*A note on the design choice.* An alternative treats capabilities as ordinary typed parameters, consistent with the object-capability principle that capabilities are just values; we separate them syntactically because "what a node transforms" and "what authority it holds" serve different review concerns, and because capabilities bind at construction while data flows at invocation.

**Edges** are typed data connections between nodes. An edge from node A's output to node B's input is valid only if the types match. Edges carry data; capabilities are not wired through edges

but provisioned via `with` clauses. This is a deliberate restriction. Object-capability systems in the E lineage permit dynamic delegation by passing capability references through messages, which is flexible but defeats static analysis of the authority topology. The signal graph trades that flexibility for a fully statically analysable capability distribution; the distributed extension of this trade-off is a separate problem discussed in Section 7. The `with` clauses collectively constitute the architecture's security policy, expressed as reviewable graph structure rather than prose. Because the graph's parameter list declares all external dependencies, swapping a production capability for a mock (replacing a live `DBHandle` with a test fixture, for example) requires only a change at the graph boundary — no node signature changes, since the `with` clause names a type, not a specific instance.

**Trust annotations** (the type-level markers introduced as trust tainting in Section 2.1) propagate through the graph. Data entering from untrusted sources carries a type marker, `Untrusted<T>`, that is preserved through transformations until explicitly discharged. In a well-typed realisation, the type system would prevent `Untrusted<T>` from reaching a node that accepts only `T`. Discharge is most effective when it is not merely a label removal but a *type transformation*: converting unstructured input into a constrained representation whose structure limits what downstream nodes can receive. The combination of trust propagation and structural typing is what would deliver the security properties claimed in Section 2.6.

An open design question must be acknowledged here: local node typing is necessary but not sufficient. The full guarantee requires the *wiring* to be checked too — an untrusted source must not reach a node expecting a clean `T` through a widening coercion — which is the standard coercion problem in information-flow type systems [30, §3]: the subtyping relation between `Untrusted<T>` and `T` must be absent, or equivalently, wiring checks must be flow-sensitive in trust. The literature offers well-understood tools — security labels in the style of Jif [32], or a lattice with no upward coercion — and the demonstrator implements a first realisation of the latter (Section 3.2). What remains open is the choice among two-point, graded (`Untrusted ⊑ Sanitised ⊑ Trusted`), and decentralised-label designs, and, critically, a soundness argument that well-typed wiring implies noninterference, which nothing here attempts (Section 7). The claim is not that the problem is solved but that it is tractable and belongs in the type system.

**Behavioural contracts** are attached to node signatures as pre- and postconditions. These are the specifications against which AI-generated implementations are verified, and the stable interface across which different implementations are interchangeable. The contract language is a Phase 1 design obligation: candidates range from refinement types in the Liquid Haskell tradition, through Dafny- or Ada-style declarative specifications, to lightweight examples-and-invariants in the QuickCheck tradition. Contracts are authored alongside node signatures — either by the developer at graph-review time, or proposed by the agent during intent capture and confirmed on review — and checked by a combination of static analysis (for properties the type system expresses directly) and property-based or contract-based testing at implementation time (for properties that require runtime evidence). The tractability trade-off between contract expressiveness and automatic verification is addressed in Section 7.

## 2.3 A concrete graph

The following pseudocode sketches an AI customer support agent as a signal graph. This scenario was chosen because it is a domain where the security properties of the signal graph model are most immediately visible: untrusted user input, LLM invocations with and without tool access, and fine-grained capability distinctions are all present. Unlike the simplified two-node illustration in Section 2.1, this example shows a realistic pipeline with structured input

parsing and content moderation. The `SanitisedPrompt` of that earlier sketch is expanded here into the refined type `ModeratedQuery`, with trust discharge decomposed across two dedicated nodes (parsing and moderation) rather than collapsed into a single transformation. No concrete syntax has been designed; this is illustrative of the kind of artifact a developer would author and review.

*Syntax conventions used below.* Types are written angle-bracketed: `T<...>`. Capability types are parameterised by scope: `DBHandle<'knowledge-base', read>` denotes a handle to the `knowledge-base` database in read mode, where `read`, `append`, and `read-write` denote modes in a permission lattice (a handle with a wider mode may be supplied where a narrower one is required: `read-write` covers both `read` and `append`, which are mutually incomparable). `LLMClient<inference>` is an inference-only LLM client (model access without tool-calling); `LLMClient<[lookup]>` grants a single named tool, `lookup`. Sum types may be written with named role labels, as in `ok: ModeratedQuery | violation: PolicyViolation | escalation: EscalationRequest` — the label on the left of the colon is an edge-addressable role, the type on the right is the variant's value type. An edge addresses a specific variant of an upstream node's output as `Node.role` (for example, `ModerateContent.ok` is the `ModeratedQuery`-valued output). These conventions stand in for a concrete syntax that Phase 1 must design; the value here is the structural shape, not the notation.

```
graph CustomerSupport(
  CustomerRequest,
  DBHandle<'knowledge-base', read>,
  LLMClient<inference>,
  LLMClient<[lookup]>,
  ResponseChannel<user-session>,
  EventEmitter<'support-queue'>
) {
  node ReceiveMessage :
    (CustomerRequest)
    → Untrusted<RawMessage>

  node ParseMessage :
    (Untrusted<RawMessage>)
    → CustomerQuery
    with LLMClient<inference>
    # discharges trust

  node ModerateContent :
    (CustomerQuery)
    → ok: ModeratedQuery | violation: PolicyViolation | escalation: EscalationRequest
    with LLMClient<inference>

  node FetchContext :
    (ModeratedQuery)
    → ConversationContext
    with DBHandle<'knowledge-base', read>

  node GenerateResponse :
    (ConversationContext)
    → ok: AgentResponse | error: LLMError
    with LLMClient<[lookup]>, DBHandle<'knowledge-base', read>

  node SendReply :
```

```
    (AgentResponse)
    → DeliveryConfirmation
    with ResponseChannel<user-session>

  node HandleLLMError :
    (LLMError)
    → DeliveryConfirmation
    with ResponseChannel<user-session>

  node NotifyUser :
    (PolicyViolation)
    → DeliveryConfirmation
    with ResponseChannel<user-session>

  node EscalateToHuman :
    (EscalationRequest)
    → EscalationTicket
    with EventEmitter<'support-queue'>

  // Data flow
  edge ReceiveMessage             → ParseMessage
  edge ParseMessage               → ModerateContent
  edge ModerateContent.ok         → FetchContext
  edge ModerateContent.violation  → NotifyUser
  edge ModerateContent.escalation → EscalateToHuman
  edge FetchContext               → GenerateResponse
  edge GenerateResponse.ok        → SendReply
  edge GenerateResponse.error     → HandleLLMError
}
```

The graph's parameter list declares its complete external dependencies: an inbound request, a database handle, two LLM clients with different permission levels, a response channel, and an event emitter. `CustomerRequest` is the domain entry type: a narrowed representation of HTTP POST traffic to `/customer/*` that can be produced either by a direct HTTP adaptor in a standalone deployment or by the `RouteRequest` dispatcher in the composed deployment shown below. Using a domain name for the entry type rather than a raw `HTTPRequest<...>` lets the same graph serve both deployments without signature churn. This list is the system's authority manifest. In production, these parameters are bound to real infrastructure; in testing, they are replaced with mocks or deterministic fixtures. Because the graph has a typed signature (its parameter list and output types), it can itself be used as a node in a larger graph. Hierarchical composition falls out naturally from the model. To make this concrete, the following graph sketches a platform that composes `CustomerSupport` (the graph above) alongside two other services:

```
graph SupportPlatform(
  HTTPRoute<'platform:*'>,
  DBHandle<'knowledge-base', read>,
  DBHandle<'billing', read-write>,
  DBHandle<'audit', append>,
  LLMClient<inference>,
  LLMClient<[lookup]>,
  ResponseChannel<user-session>,
  ResponseChannel<agent-session>,
  EventEmitter<'support-queue'>
) {
```

```
  node RouteRequest :
    (HTTPRoute<'platform:*'>)
    → customer: CustomerRequest | agent: AgentRequest | billing: BillingRequest

  node CustomerSupport :
    (CustomerRequest)
    → DeliveryConfirmation | EscalationTicket
    with DBHandle<'knowledge-base', read>, LLMClient<inference>, LLMClient<[lookup]>,
ResponseChannel<user-session> @customer_session, EventEmitter<'support-queue'>

  node AgentDashboard :
    (AgentRequest)
    → DeliveryConfirmation | EscalationTicket
    with DBHandle<'knowledge-base', read>, ResponseChannel<agent-session>

  node BillingService :
    (BillingRequest)
    → DeliveryConfirmation | EscalationTicket
    with DBHandle<'billing', read-write>, ResponseChannel<user-session>
@billing_session

  node RecordAudit :
    (DeliveryConfirmation | EscalationTicket)
    → AuditConfirmation
    with DBHandle<'audit', append>

  // Data flow
  edge RouteRequest.customer → CustomerSupport
  edge RouteRequest.agent    → AgentDashboard
  edge RouteRequest.billing  → BillingService
  edge CustomerSupport       → RecordAudit
  edge AgentDashboard        → RecordAudit
  edge BillingService        → RecordAudit
}
```

`CustomerSupport` is no longer nine nodes visible at this level; it is a single node with a typed signature. Its internal wiring — the trust zones, the graduated LLM access, the moderation routing — is encapsulated. At the platform level, the reviewer sees only what authority each service holds and how data flows between them. The platform's parameter list is the union of its sub-graphs' requirements: the `DBHandle`, `LLMClient`, and channel capabilities each appear exactly where they are needed. `BillingService` has read-write access to the billing database but no LLM access; internally it would be a conventional pure pipeline — input validation, policy checks, and DB transactions — with no reliance on LLM nondeterminism, and how its shape differs from `CustomerSupport` is itself visible at the composition level through the absence of LLM capabilities. `AgentDashboard` can read the knowledge base but cannot write to billing. These constraints are visible at a glance in the capability annotations.

The boundary output `ServiceOutcome` declared here is the aggregation of `CustomerSupport`'s internal terminal outputs — in this example, the disjoint union of `DeliveryConfirmation` (from the reply paths) and `EscalationTicket` (from the escalation path). The sub-graph has several terminal nodes of different types, and the composition requires a single boundary type; treating `ServiceOutcome` as that aggregated union is the simplest of several possible designs. The same type also serves as the boundary output of `AgentDashboard` and `BillingService`: at the platform level, `ServiceOutcome` is the shared outcome currency that `RecordAudit` consumes, and each

service maps its own terminal types onto it. How multi-terminal sub-graphs should expose their outputs (as a union-typed boundary, as named multi-output ports, or via an explicit aggregation node inside the sub-graph) is a Phase 1 language-design question noted in Section 7.

Hierarchical composition also surfaces a design obligation this work does not claim to have solved. When `SupportPlatform` provisions `LLMClient<inference>` to `CustomerSupport`, the sub-graph's internal wiring must route that capability to the specific internal nodes that require it (`ParseMessage` and `ModerateContent`) and not to others. How a sub-graph exposes its internal capability requirements at its boundary — as a flat union reabsorbed by convention, as named capability slots, or via structural matching on capability types — is an open question noted in Section 7 and part of the Phase 1 language design. The complementary, narrower question — which *instance* of a capability type a parent routes to each sub-graph — is now expressible in the canonical graph source: `CustomerSupport` and `BillingService` both require `ResponseChannel<user-session>`, and the platform binds each a distinct named instance (`customer_session` and `billing_session`, shown in the composition figure below), so identity is carried across the composition boundary rather than aliased by type. Section 3.4 and Section 3.5 report what the demonstrator makes of both questions: the narrower one is implemented and tested, the internal-fan-out surface remains open.

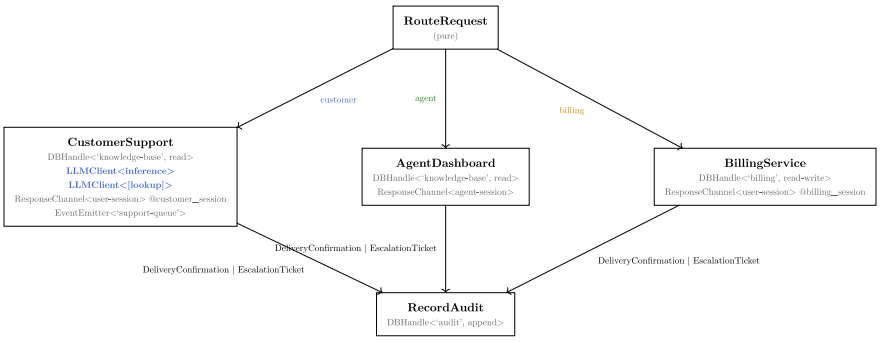


Figure 2: The SupportPlatform composition graph. Each service node is a sub-graph (the nine-node CustomerSupport graph is now a single node). Capability annotations show the authority distribution at the platform level; where a service holds a distinct named instance of a capability type, the identity label is shown after `@` (the two services holding `ResponseChannel<user-session>` are routed distinct `customer_session` and `billing_session` instances). The audit node collects outcomes from all services with append-only database access.

The `CustomerQuery` type is central to the security argument. `ParseMessage` does not merely strip the `Untrusted` wrapper from free text: it transforms unstructured input into a constrained representation — a classified intent (from a finite set of categories), extracted entity references, and bounded text fields — whose type limits what downstream nodes can receive. The raw message is consumed; downstream nodes never see it. `ModerateContent` then refines this further into a `ModeratedQuery` (the `.ok` variant of its output), a type distinct from `CustomerQuery` that records, at the type level, that a moderation check has occurred. Downstream nodes accept only `ModeratedQuery`, so a wiring that bypasses moderation is ill-typed; the type distinction makes the moderation step *load-bearing* rather than advisory.

This is a stronger guarantee than trust annotation alone: a well-typed `ModeratedQuery` cannot carry arbitrary executable instructions in positions that flow to privileged nodes, *provided the schema is designed to exclude unbounded free text in those positions.* That proviso is essential: the defence is layered, not absolute, and a schema that keeps a free-text field — most do, for

the original question itself — leaves adversarial data in it. Section 4.3 measures exactly this residual on the built graph. Each layer is nonetheless visible in the graph topology and, in a sound realisation, enforceable by the type system.

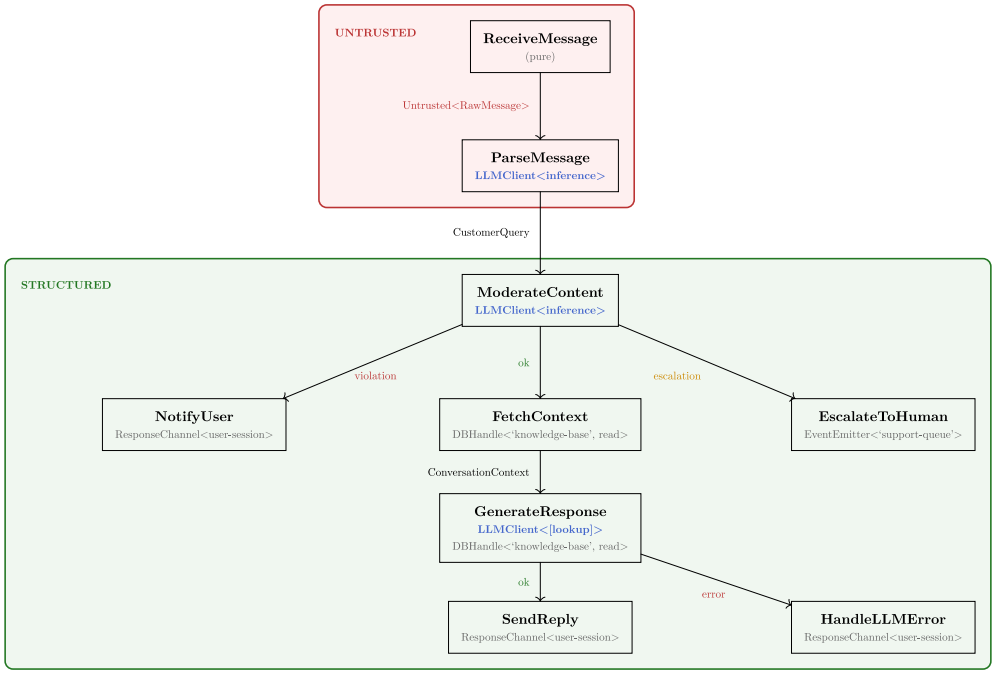


Figure 3: The CustomerSupport signal graph. Red shading marks the untrusted zone; green shading marks the structured region. Edges show data flow; capability requirements are annotated on each node. LLM access (blue) is graduated: inference-only for parsing and moderation, a single lookup tool for response generation.

This diagram is simultaneously the architecture model, the security policy, and the program. Several properties are visible at a glance, without reading any implementation code.

*Prompt injection is addressed through structural typing and topological constraint.* `ParseMessage` transforms raw input into a `CustomerQuery`, a constrained representation that discards the original free text. It is worth naming the trust placement this entails: the LLM is itself the trust-discharging component, since `ParseMessage` consumes `Untrusted<RawMessage>` and emits a non-`Untrusted` `CustomerQuery` on the strength of the LLM's classification. This trust is bounded by the LLM's capability shape (`LLMClient<inference>` grants model access without tool use) and by the schema of `CustomerQuery`, which constrains downstream exposure regardless of how the LLM is influenced. Both LLMs that process user input (`ParseMessage` and `ModerateContent`) have `LLMClient<inference>`: even if adversarial instructions influence their behaviour, they have no mechanism to act on them. The tool-capable LLM (`GenerateResponse`) receives only `ConversationContext` assembled from a `ModeratedQuery` and knowledge-base lookups — never raw user text. A direct path from untrusted input to a tool-capable LLM does not exist in the graph; in a sound realisation of the type system, it would be ill-typed. A subtlety must be acknowledged: capability restriction prevents the LLM from *acting* on adversarial instructions, but not from being *influenced* in its classification. An adversarially crafted message could cause `ParseMessage` to produce a `CustomerQuery` that misclassifies intent, routing the query to the wrong downstream path. The defence here is twofold: the attack surface is narrowed from arbitrary tool execution to incorrect routing within a typed pipeline (a qualitative reduction in severity), and the routing itself operates within the user's authorised scope — if downstream

actions are bounded by the user's own credential-scoped capabilities (passed as a parameter at the graph boundary), a misrouted query can only trigger actions the user was authorised to perform. Separating user-level authorisation from node-level capability injection is a design consideration for Phase 1 and is listed as an open item in Section 7.

*Capability distribution is minimal and visible.* The `with` clauses are a complete manifest of the system's authority. `ReceiveMessage` is pure: it transforms a typed customer request into a domain message with no `with` clause. `ParseMessage` and `ModerateContent` have inference-only LLM access: enough to classify and evaluate text, not enough to act on instructions. `FetchContext` has read-only knowledge-base access. `GenerateResponse` has a scoped LLM client with a single lookup tool and read-only database access. Terminal nodes (`SendReply`, `NotifyUser`, `HandleLLMError`) each have only a session-scoped response channel. `EscalateToHuman` can emit to the support queue but cannot read or write any database; its `EscalationTicket` output is a published identifier that a containing graph may route onward (for auditing or agent-dashboard display) or discard, consistent with the pattern that every node produces a typed value whether or not the current composition consumes it. No node has more authority than its function requires.

*Conditional routing and error handling are explicit.* `ModerateContent` produces a three-way union: approved queries continue to the response pipeline as `ModeratedQuery`, policy violations are routed to user notification, and ambiguous cases are escalated to human agents. `GenerateResponse` returns `AgentResponse | LLMError`, with the error case routed to `HandleLLMError`. In both cases, the routing is a structural property of the graph, visible in the diagram and the pseudocode. The validator enforces that every declared variant has a downstream consumer, but it cannot force a node's implementation to emit the *correct* variant for a given input — a `ModerateContent` that always emits `.ok` would be well-typed but behaviourally wrong. Variant-correctness is a behavioural-contract obligation, checked by the shallow-verification work of Section 7.

The example elides transient infrastructure failures (a classification failure in `ParseMessage`, a read error in `FetchContext`); a production-shape graph would route those into error variants by the same mechanism shown for `GenerateResponse`. The precise syntax for conditional routing, fan-out, and error propagation is an open Phase 1 design question (Section 7.1).

## 2.4 The development workflow

The workflow below is proposed rather than implemented: the demonstrator covers the structural half of its verification step and none of the rest (Section 5.3).

1. **Intent capture.** A human describes a change in natural language; an SDD-style tool translates it into a proposed graph transformation — new nodes, modified `with` clauses, changed signatures or contracts. Extending existing SDD frameworks from prose specifications to *typed graph transformations* is the step most load-bearing for the workflow to cohere, and a core Phase 1 deliverable (Section 7.1).
2. **Graph review.** Humans review the diff as a visual graph change — new nodes, new capability requirements, and trust-boundary crossings flagged — which is simultaneously an architecture, security, and design review of a typed program transformation rather than of prose.
3. **Implementation.** AI agents generate each node's code in isolation, from its signature, contracts, and input/output types, with no visibility into adjacent implementations. Isolation is not about less context helping in general — [9] shows the opposite — but about forcing

every cross-node dependency through the typed signature, so the graph stays the complete specification of inter-node interaction; anything an agent would otherwise glean from neighbours (error conventions, retry shapes) must be lifted into types or contracts to have effect.

4. **Verification.** Automated tooling confirms implementations satisfy their contracts, the assembled graph conforms to its declared types, and no node exceeds its injected capabilities — primarily structural checking, since the runtime enforces the capability restrictions.
5. **Merge.** On passing verification the transformation merges: the human approved the graph diff, the machine confirmed conformance. Architectural and security properties are decided at the graph level, so contract-satisfying implementations are accepted without line-by-line review, spot checks aside.

Nodes are individually testable by injecting mock capabilities and asserting outputs against input sequences; graph-level tests are expressed the same way at the boundary. The replay mechanism of Section 2.1.3 would double as a regression suite.

## 2.5 The runtime

In the intended runtime, each node would execute in a lightweight, capability-restricted sandbox (a WASM module, a Monty-style interpreter, or a BEAM-like process), with CHERI hardware where available. What the demonstrator implements is a strict subset of this — two tiers, one of them WASM components, no CHERI, no BEAM, no Monty (Section 3.3). The intended properties are:

**No ambient authority.** A node would perform no side effect beyond calling methods on its injected capability objects — no library import, filesystem, or network access — enforced by the absence of any mechanism rather than a policy guard (Section 2.2).

**Defence in depth.** The type system would forbid the graph from expressing illegal capability grants; the sandbox would stop generated code exceeding its injected capabilities; the OS would compartmentalise processes; and CHERI would prevent capability forgery at the memory level. These layers are not fully independent — the runtime's injection is configured by the type system's analysis, so a type-system bug could misconfigure it — but they fail differently: a sandbox escape does not help an attacker lacking a hardware capability, and a type-system error does not cross a correct OS compartment. Weaker than fully independent enforcement, stronger than any single layer.

**Language agnosticism.** WASM is the intended target (Rust, C, C++, Go, Python via interpreters such as Monty). Interfaces are defined in a language-neutral type system, so the implementation language is an optimisation choice — the basis for the interchangeability the confined tier demonstrates across its Rust node bodies (Section 3.3.2).

**Snapshotting and resumption.** Nodes would be pausable, serialisable, and resumable, enabling the durable execution and replay of Section 2.1.3. BEAM supports this natively; WASM support is maturing (WASI 0.3 [33] brought native async to the Component Model) but mid-execution component snapshotting is not yet solved — a Phase 1 implementation constraint, not a limitation of the model.

**Evaluation strategy.** A differential strategy — evaluating only the nodes a change affects, in the tradition of differential dataflow [26] and DBSP [28] — is the target, and is what would make per-node sandboxing cost-acceptable at scale. The demonstrator implements neither it nor naive re-evaluation, propagating signals along an active path in a single thread (Section 8.3).

## 2.6 Security properties

The capability-injection model would provide security guarantees qualitatively different from those achievable by code review or runtime monitoring.

**Injection attacks.** SQL injection and command injection depend on untrusted input reaching an interpreter in executable form. In the signal graph, a SQL-executing capability would accept typed queries, not raw strings. `Untrusted<string>` could not reach it without passing through a sanitisation node that produces a typed query. In a sound realisation of the type system, the pattern would be ill-typed — rejected by the type system rather than left to convention.

**Prompt injection.** Structurally attenuated, not eliminated: no well-typed wiring would connect an untrusted source to a tool-capable LLM without transiting a type that constrains the downstream payload, but a free-text field that reaches an LLM-capable node stays adversarial even when its wrapper is non-`Untrusted`. Strength depends on schema discipline; Section 4.3 measures the residual on the built graph.

**Supply chain attacks.** The defence is *capability scoping*, not I/O denial. A malicious library inherits only the scope of the node it inhabits — a `DBHandle` bound to one database in one mode, an HTTP client scoped to declared endpoints, an LLM client to a tool allowlist — so the blast radius is that scope, not the library's creativity. This holds as a property of the artifact on the confined tier and only as an object discipline a hostile library can reach around on the host tier (Section 4.4); on CHERI hardware even a memory-safety exploit could not forge a capability to memory it was not granted. A new capability requirement surfaces in the graph diff as a new `with` clause — a reviewable change, not an implicit elevation.

**Confused deputies.** A privileged node acting on instructions derived from an unprivileged source is the classical confused-deputy pattern; the signal graph attenuates it through two mechanisms. First, capability scoping (above) limits the damage any one node's authority can do. Second, for operations whose safety depends on *which user* initiated them, capabilities passed at the graph boundary can be bound to the calling user's credentials, such that downstream nodes operate only within that user's authorised scope. The distinction between user-level authorisation and node-level capability injection — and the precise mechanism by which the former is threaded through the latter — is a Phase 1 design obligation noted in Section 2.3 and expanded in Section 7.

**Covert channels remain an open concern.** A node granted a permitted capability can encode information into the timing, query shape, or ordering of its legitimate outputs — channels the type system does not model, and which general covert-channel elimination (a known-hard problem) is not claimed to close. The forward plan for characterising them is Section 7.2.

**Privilege escalation.** A node would not be able to acquire capabilities it was not given. The graph would be the complete and sole description of the system's capability distribution. On CHERI hardware, this guarantee would extend to the memory level. Two caveats catalogued in Section 7 bound this claim: a node holding a network capability can in principle acquire authority out-of-band from services the graph does not model (distributed ambient authority), and for user-scoped operations the guarantee is only as strong as the principal-binding design that Phase 1 must deliver.

# 3 Implementation

The demonstrator is a working, tested implementation of the graph-level analyses and the capability-injected runtime described in Section 2. It is deliberately small, and its scope should be stated before its results are: it is **not** the Phase 1 language of Section 7. It implements no noninterference proof, no dependent types, no contract language, and no visual editor. What it does implement is the part of the vision whose implementability was least obvious from the outside — that a graph carrying capability and trust annotations can be statically analysed to reject unsafe wirings, and that node bodies can be confined to exactly the authority their signatures declare — and it does so with enough tooling to be run, tested, and measured.

Everything in this section is built and tested; the present tense is used accordingly. Where a property is asserted by a test, that is said: a claim backed by a test cannot silently drift from the artifact the way prose can.

## 3.1 Canonical graphs and the type grammar

The graph definitions in `graphs/*.json` are the single source of truth, validated against a JSON Schema. Both the pseudocode listings and the SVG diagrams reproduced in Section 2.3 are generated from them, so the figures in this paper cannot drift from the graphs the validator and runtime actually consume — the two graphs shown are the two graphs run.

A small type-expression parser reads the capability-annotated type grammar: angle-bracketed generics, capability scopes and modes, and sum types with role labels. The documented grammar is itself emitted from the parser as a build artifact rather than maintained by hand, on the same principle.

## 3.2 The validator: six graph-level analyses

A dependency-free validator implements six classes of analysis over the canonical graphs.

**Edge type-compatibility** source output types, with sum-type variant resolution, must match target input types in data shape, independently of trust.

**Trust-lattice checking** trust levels form a two-point lattice `Untrusted ⊑ Trusted` with no upward coercion — `Untrusted<T>` is not a subtype of `T` — and the wiring check is flow-sensitive with respect to trust, applied uniformly to edges **and** to node bodies. A node may raise trust only if it is declared a discharger with `discharges_trust: true`, as `ParseMessage` is in the `CustomerSupport` graph.

**Variant completeness** every declared sum-type output variant must have at least one consuming edge, so dead branches are visible at the graph level rather than hidden inside node implementations.

**Capability narrowing** at cross-graph composition a parent may provide a handle whose authority exceeds what the sub-graph declares — a wider `LLMClient<[...]>` tool set, or a `DBHandle<X, read-write>` where `read` is required — while strict equality still governs every data-flow position.

**Intra-graph consistency** every declared capability is used, every boundary data input is consumed, and every edge references a valid node and output variant.

**Cross-graph signature matching** a node whose name matches another graph's name must satisfy that graph's parameter list position by position.

The second of these is the one worth dwelling on, because its design changed under implementation and the change is a result rather than a detail. Trust is **not** a rule beside edge typing; it is the same no-upward-coercion order, applied to two places. The consequence is what Section 4.1

measures: the blunt unsafe wiring — untrusted input run straight into the tool-capable node — fails as an ordinary data-shape mismatch, but the subtle one survives that check. Widen the tool-capable node's declared input to `Untrusted<RawMessage>` and every edge in the graph type-checks on data shape. The graph is still rejected, because the tool-capable node now emits a non-`Untrusted` output from an `Untrusted<_>` input without being a declared discharger, which is an upward coercion under the lattice.

That case is the argument for the discipline. An earlier design that folded trust into ordinary subtyping — admitting `Untrusted<T> <: T` — would have accepted the widened graph, with the failure invisible at the level of edge types. Trust laundering is thereby one violation of one principled order, and local edge data-compatibility is shown not to be the property worth checking. What the validator does **not** establish is soundness: that every graph well-typed under this lattice satisfies noninterference. That obligation is untouched, and is Phase 3′s (Section 7.3).

## 3.3 Two enforcement tiers

The runtime loads the same canonical graph JSON, instantiates each node with injected capability handles, and propagates signals along the active path. Nodes run on one of two enforcement tiers, and the runtime reports which ran each node. The distinction between them is the whole point, so it is stated precisely.

### 3.3.1 The host tier

The default tier is **host-level object discipline**, not unforgeable containment. Each node receives only the handles its signature declares, and each handle exposes only the operations its type permits: a handle for `LLMClient<inference>` has no tool-calling operation at all, and one for `LLMClient<[lookup]>` refuses every other tool. This demonstrates the *shape* of the guarantee. It does not enforce it against a node that declines to play by the object model — nothing stops a hostile Python node from importing `os` and reading the filesystem directly.

That gap is not left as a caveat in prose. A hostile-node suite asserts that filesystem, network, environment, and ungranted-capability escapes **succeed** on this tier. The gap is recorded as a passing test, which is the only form in which it cannot be forgotten, and it is reported in Section 4.4 rather than omitted from the evaluation.

### 3.3.2 The confined tier

The second tier closes that gap for the nodes ported to it. Node bodies are compiled to WebAssembly **components** (Section 6.9) — Rust, via `wit-bindgen`, to `wasm32-unknown-unknown`, converted to a component with `wasm-tools` — and run under `wasmtime`. Each capability kind is declared as a typed WIT interface: `LLMClient<inference>` becomes an interface offering a single inference function, `DBHandle<'knowledge-base', read>` one offering `lookup` and no writer. A node's `with` clause is then precisely the set of interfaces its component imports.

On this tier the inference-only guarantee changes in kind rather than degree: an `LLMClient<inference>` node has no tool-calling **import**, not merely no tool-calling method, and a component declaring a capability it was not granted cannot be instantiated at all — the refusal lands before any guest code runs. Two properties follow from the boundary being typed rather than byte-oriented, and both are asserted by tests rather than argued in prose.

**There is no ambient authority to reach for, as a property of the artifact rather than of the host's configuration.** An earlier iteration of this tier compiled nodes to `wasm32-wasip1` modules run under an empty WASI 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. Compiled for `wasm32-unknown-unknown` and linked with no WASI adapter, no filesystem, socket, environment, or clock function appears among the imports at all — the hostile component imports nothing whatsoever. What was a configured absence is now a structural one, and the suite asserts the import set directly. A `wasip2` build would reintroduce the problem through `wasi:cli/*`, which is why the tier targets a bare component rather than a WASI snapshot.

**The boundary cannot silently drift from the node's declared signature.** Because a component's imports are typed interfaces rather than opaque symbols, the import set the runtime will accept is **derived from the graph**: each node's expected interfaces are computed from its `with` clause in the canonical JSON and compared against what the built component actually imports. A world that grants a node an interface its signature never requested fails that check rather than shipping as a quiet over-grant.

The same typing absorbed two node-body checks into types. Under the earlier flat `(ptr, len)` ABI, `ParseMessage` received an intent label as a string and had to **check** it against the permitted set so that adversarial text could not widen it; expressed as a WIT `enum`, widening is not representable, the check is gone, and the property is held by the type rather than by the node's own diligence. The model's reply is likewise a WIT `variant`, so a malformed reply has no encoding.

On the demonstrated customer path, 5 of its 6 nodes run as components — every node but the pure `ReceiveMessage` narrowing — while the platform around them runs on the host tier, so the two tiers demonstrably compose within one graph (the incremental-migration path of Section 7.2, from opaque host node to confined node) and a majority of the taken path is confined rather than two isolated nodes. Each of the five runs from a Rust body regenerated from the same signature and contract as its Python counterpart, with the graph unchanged — a concrete instance of the code-as-compiled-artifact property, the same contract with a different implementation language. Crucially the set now spans the typed capability boundary and not only pure transformations: `FetchContext` holds a read-only knowledge-base handle, `GenerateResponse` a scoped tool set plus that handle, and `SendReply` an identity-routed send whose single crossing lands on exactly the channel instance the parent routed, the confirmation carrying that instance's identity back across the WIT boundary. What interchangeability across languages then means, and what bounds it, is taken up in Section 5.

What this tier does not provide is memory-level unforgeability. Enforcement is unforgeable at the WASM boundary; a sandbox escape via a runtime defect is out of scope, and hardware-backed enforcement (CHERI) remains Phase 3 work. Nor does typing the boundary touch the residual free-text channel — Section 4.3 reports that residual, measured on this tier on purpose.

## 3.4 Capability identity, revocation, and rotation

The vision named capability revocation as an open problem and, in doing so, exposed a prerequisite it had not noticed: revoking a **specific** handle first requires a way to **name** it. With capabilities shared by type, there is no instance to revoke without severing every node that names the type.

The runtime's default is indeed to provision one handle per declared capability **type**, shared across every node whose `with` clause names it, so two nodes each declaring `DBHandle<'knowledge-base', read>` receive the same object. For read-only handles this is harmless; for stateful,

rate-limited, or revocable handles it is not. Capability **identity**, not merely capability type, must therefore be expressible at the graph boundary.

It now is, and in the graph source rather than only through an assembly API: each node may carry a `capability_identities` map binding a declared capability type to an identity label. A capability with no identity declared keeps the type-shared default, so existing graphs are unchanged. Because identity lives in the JSON, the validator checks it — an identity declared for a capability a node does not hold is rejected at validation time — and the generated pseudocode and diagrams render it, so a distinct instance is visible in the artifact and cannot drift from the source. That two same-typed handles carry independent state is a tested property.

On that prerequisite, revocation and rotation are built as one mechanism. An instance provisioned as **revocable** is wrapped in a caretaker [29]: the node holds a forwarding proxy, and a separate revoke authority — kept by the host that assembled the graph, never handed to a node — severs it, after which the node's next use fails loudly instead of exercising authority. **Rotation** re-points that same proxy at a new resource behind the same identity, guarded to a same-kind replacement so the surface the node holds cannot change kind underneath it. Revoke and rotate are two levers on one indirection rather than two mechanisms. Both are **targeted** (severing one identity leaves its same-typed siblings, and the type-shared default, untouched) and **opt-in** (an instance not declared revocable is provisioned bare), and both are tested.

Enforcement reaches the confined tier as well: a sandboxed node reaches its capabilities through typed WIT host functions backed by the very same caretaker, so severing it makes the guest's next capability crossing fail at the boundary — and there it is unforgeable, because the confined node's import set is its whole world and it has no other path to the resource. On the host tier the caveat is unchanged: severing binds the handle's authority, but not a node that reaches around the object model. Sandbox-tier **rotation**, and the graph-transformation form of both operations, remain undesigned (Section 7.2).

## 3.5 Operational composition: sub-graph execution

Hierarchical composition is operational, not merely structural. A node whose name resolves to another graph is a **sub-graph node**, which the runtime executes by nested assembly and execution: the same assembly gate, the same executor. Composition therefore adds no second execution model — a sub-graph is a node whose body happens to be a graph.

`SupportPlatform` accordingly runs rather than merely assembles. A customer request entering at its `HTTPRoute<'platform:*'>` boundary is dispatched by `RouteRequest`, crosses into the nine-node `CustomerSupport` graph, and its outcome crosses back to `RecordAudit`, which appends it to the audit log through an append-only handle that cannot read the log back. Four properties of that crossing distinguish composition from mere call nesting, and all four are tested on the shipped graph.

1. **A sub-graph cannot provision authority of its own.** The executor holds no backend to provision from, so the child receives exactly the handles the parent routed and nothing else. Capability confinement across the composition boundary is a property of the plumbing rather than a rule to be remembered — the case we most expected to need enforcing turned out not to need it, because there was no mechanism by which it could fail.
2. **Identity routing has an executable consequence.** The distinct `customer_session` instance the platform declares for `CustomerSupport` is the instance the reply node **inside** that sub-graph sends on, and its sibling's instance is untouched.

3. **The parent matches the sub-graph's flat parameter list by position and type.** This is option (i) of the capability-routing question the vision left open; it is now settled at the runtime, while the fuller slot surface remains a language-design choice (Section 7.1).
4. **Composition crosses enforcement tiers.** A host-tier `SupportPlatform` runs `CustomerSupport` with its nodes on the confined tier, the child's nodes resolving to their own tiers exactly as in a top-level run. This needs no new mechanism, which is itself the observation: because the executor holds no backend, confinement across the boundary holds unchanged whichever tier a child node runs on, and severing a routed instance at the parent makes the confined crossing fail two altitudes down — nothing on the child side can re-mint the severed capability.

The demonstration is bounded, and the bounds are worth stating rather than leaving to be discovered. Only leaf nodes on the taken path have implementations: `AgentDashboard` and `BillingService` are neither defined as graphs nor registered as node bodies, so those branches do not run. A sub-graph run reaching several terminals has no single boundary value, and the runtime **refuses** it rather than electing a winner — the multi-terminal aggregation question stays open, and fails loudly instead of silently.

The output side of a sub-graph's signature is now checked, closing a gap the vision left open. `ServiceOutcome` is the union alias of the sub-graph's terminal types (`DeliveryConfirmation | EscalationTicket`) — the convention Section 2.3 adopts — and the earlier demonstrator asserted that name in the JSON while checking it against nothing: the cross-graph analysis related a sub-graph node's **inputs** to the referenced graph's parameters and never examined the output side, so a sub-graph node could declare any output type whatsoever and no check would object. The fix is **structural, not nominal**: the graph language still has no alias mechanism, so rather than assert an unresolvable name the canonical graph spells the union, and the cross-graph analysis requires a sub-graph node's declared boundary output to equal the union of the referenced graph's terminal output types. A node narrowing that union — claiming only `DeliveryConfirmation` and hiding the escalation path — is now rejected at validation, with every edge still type-checking, so the fault is caught by the output-side check rather than as an edge type mismatch; a mutation-corpus case pins that verdict and its reason class.

## 3.6 The execution trace

Every run now emits a structured, machine-readable **trace**: per node in execution order, the enforcement tier that ran it, the trust labels of its input and output, and the distinct capability crossings it made — each recorded as the typed interface crossed and the capability instance it landed on. Sub-graph runs nest, so a composed execution is inspectable at both altitudes. The format is pinned by a committed JSON Schema (`poc/trace-schema.json`) and its structure is deterministic: two runs of the same graph and input serialise identically, because nothing structural carries a timestamp and timing lives in one optional field excluded from comparison. This makes the run itself a checked artifact in the repository's established pattern — a rendered output generated from a checked source — and it is the artifact the planned graph inspector consumes, which keeps that interface a pure renderer rather than a second source of truth.

A crossing is recorded as `(interface, instance)` and deduplicated per node, not as a per-call log, and that choice is load-bearing for a claim the two tiers would otherwise only appear to satisfy. The host tier calls a capability method once; the confined tier's component runs its own internal loop, crossing the same interface a different number of times under a different function name. Recording the *set* of interfaces a node crossed, and the instance each landed on, is the one representation both tiers produce identically — the host tier at the injected handle

wrappers, the confined tier where the guest calls back across the WIT boundary through those same handles — and a test asserts the two traces are structurally equal once the tier field, the one thing that legitimately differs, is set aside. Instance attribution is what makes identity routing observable in the trace rather than only at assembly: the crossing the reply node makes inside `CustomerSupport` is recorded against the `customer_session` instance the platform routed, not the bare channel scope it would carry standalone. On the confined tier a node body cannot forge or suppress an entry, because recording sits at the boundary it cannot reach around; on the host tier recording is exactly as circumventable as every other host-tier guarantee, and the trace claims nothing stronger there.

The evaluation harness emits canonical traces of the prompt-injection scenario on both tiers into `dist/`, and pins their structural properties in the same build-failing style as the corpus verdicts. Two properties are pinned. First, trust is raised only at the declared discharge node (`ParseMessage`): a node that turned an untrusted input into a trusted output anywhere else would fail the build. Second — and this is the free-text residual of Section 4.3 made visible in data rather than only in prose — the untrusted taint reaches the tool-capable node (`GenerateResponse`) through a *permitted* field even on the confined tier: the node runs confined and its input is labelled *trusted* (the `Untrusted<_>` wrapper was discharged upstream), yet adversarial text is still present in the question field it received. Pinning this keeps stronger enforcement from being misread as a stronger claim; the guarantee remains attenuation, not elimination.

One boundary is worth drawing sharply, because the trace sits next to a prediction it does not fulfil. Section 2.1.3 argues that under the runtime's purity guarantees every capability crossing would be a typed, observable event whose structured log would make past behaviour substantially reproducible — replay-and-fork debugging, a regression suite authored for free. The trace is that structured log of crossings, and having it is a genuine prerequisite for replay. 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. Replay was not attempted (Section 5.3), and this artifact does not move that verdict; it substantiates the narrower, adjacent fact that the crossings the vision would journal are the crossings the runtime already sees.

# 4 Evaluation

This section evaluates the demonstrator along four dimensions: which unsafe wirings the static analyses reject and by which analysis, what a capability-boundary crossing costs against the envelope the vision asserted without evidence, what an adversarial message can reach once it is inside the graph, and what each enforcement tier actually stops.

**Method.** Every figure in this section is interpolated from a build artifact emitted by a single run of the evaluation harness, which imports the mutation corpus, the benchmark, and the injection scenario rather than re-deriving them. No number below is transcribed by hand. The harness is a regression guard rather than a report: each corpus case pins the verdict it must produce **and the reason class it must be caught by**, and any divergence is rejected instead of rewriting the table. Pinning the reason matters more than pinning the verdict — a pass/fail pin would stay green if trust laundering silently began to be caught as an ordinary type mismatch, which is precisely the regression that would hollow out the central claim. The corpus also pins itself against silent growth: adding an unsafe variant without pinning it is an error, not an uncounted pass.

**Scope.** The corpus is **curated and illustrative**. It contains the unsafe wirings we thought to write down, and the counts below report how many of those were caught. That is evidence the graph-level analyses are implementable and catch the mistakes they target; it is not a soundness result, and no claim is made that the corpus is exhaustive or that an uncaught class does not exist. Section 8 states this and the other limits in full.

The overhead figures are wall-clock timings from one machine, reported to establish an order of magnitude rather than as portable benchmarks. They were produced on `Linux-6.17.0-1020-azure-x86_64-with-glibc2.39` (`x86_64`), Python `3.12.3`, wasmtime `46.0.1`.

## 4.1 Graph-mutation corpus

Each case is a graph the validator either accepts or rejects at assembly time, before any node executes. The canonical graphs must be **accepted**: a validator that rejected everything would catch every unsafe wiring and be worthless, so the safe cases are corpus entries in their own right rather than a formality.

| case | kind | verdict | caught by | |
|---|---|---|---|---|
| `customer-support` | canonical | accepted | — | ✓ |
| `support-platform` | canonical | accepted | — | ✓ |
| `bypass_pipeline` | mutation | rejected | edge type-compatibility | ✓ |
| `launder_trust` | mutation | rejected | trust lattice | ✓ |
| `mislabel_subgraph_output` | mutation | rejected | cross-graph signature | ✓ |

Table 1: The graph-mutation corpus. Each unsafe case is pinned to the analysis that must catch it, not merely to the fact of rejection.

2/2 safe wirings were accepted and 3/3 unsafe wirings rejected at assembly time, each by its pinned reason class.

The two mutations are chosen to separate a blunt mistake from a subtle one. `bypass_pipeline` wires untrusted input straight into the tool-capable node; the edge simply does not type-check on data shape, and any structural checker would catch it. `launder_trust` is the interesting case: it **repairs** that mismatch by widening the tool-capable node's declared input to `Untrusted<RawMessage>`, after which every edge in the graph is data-compatible. It is nonetheless rejected, as an upward coercion — the node raises trust without being a declared discharger — and it is rejected by the trust lattice rather than by edge typing, which is the property the pin enforces.

This is the corpus's whole point, and its limit. It establishes that the distinction between "type-checks" and "is well-trusted" is real, implementable, and load-bearing on a graph that would otherwise pass. It does not establish that no third kind of mistake exists.

## 4.2 Capability-boundary overhead

The vision asserted a working envelope it had no numbers for: if a node performs on the order of 10 ms of useful work, a per-crossing cost below roughly 1 ms keeps total overhead under about 10%. These are the numbers, whichever way they fall.

| measurement | value | notes |
|---|---|---|
| component compilation | 11.59 ms | one-time per artifact, cached; not paid per call |
| instantiation (per node) | 0.078 ms | the tier's fixed per-node price; pooling would amortise it |
| `ParseMessage` run (warm) | 0.104 ms | 1 crossing |
| `GenerateResponse` run (warm) | 0.275 ms | 3 crossings |
| **marginal per-crossing** | **55.6 µs** | differenced between two warm paths through one component |

Table 2: Capability-boundary overhead on the confined tier. Every crossing lifts and lowers typed WIT values.

A crossing costs 55.6 µs — within the 1 ms envelope, projecting to 0.56% overhead on a node doing 10 ms of work. The measurement was worth making because the typed boundary is not obviously free: the tier's first implementation passed a packed `(ptr, len)` pair into linear memory and let both sides parse the bytes, whereas every crossing measured here marshals typed records, enums, lists, and variants. Typing the boundary did not cost the performance argument.

Two cautions bound the reading. First, these figures should **not** be compared like-for-like with those previously reported for the flat-ABI tier, because the benchmark's timing discipline was corrected at the same time: it now warms each measurement and takes the best of several rounds, where before it timed a single cold pass and so charged JIT and interpreter warm-up to whatever it measured first. Since the per-crossing cost is derived by **differencing** two timings, that defect could distort it by an order of magnitude in either direction. The comparison that survives is the one that matters: typing the boundary did not move the crossing out of its order of magnitude. Second, these are proof-of-concept figures from a Python host and a two-node slice, not a production runtime at graph scale. They establish that the crossing itself is cheap and locate the fixed cost in instantiation; they say nothing about serialisation cost for complex types, or about graphs with hundreds of nodes.

## 4.3 Prompt-injection attenuation

An adversarial message instructing the model to call an `exfiltrate` tool is driven through the `CustomerSupport` graph. What matters is not whether the model is fooled — assume it is — but what it can **reach** once fooled. The message traverses 6 nodes: `ReceiveMessage` → `ParseMessage` → `ModerateContent` → `FetchContext` → `GenerateResponse` → `SendReply`, of which 5 run on the confined tier and the rest on the host tier.

| property | result | |
|---|---|---|
| the tool-capable node received | `ConversationContext` | ✓ |
| it received an `Untrusted<_>` value | `false` | ✓ |
| an out-of-scope tool call was refused | `true` | ✓ |
| adversarial text still present in a permitted field | `true` | — |

Table 3: Prompt-injection outcome. The last row is the residual, and it is asserted on the **confined** tier deliberately.

**The residual, stated plainly.** The last row is not a failure; it is the boundary of the claim. The `Untrusted<RawMessage>` value is consumed at the parse boundary and never reaches the tool-

capable node, which receives a `ConversationContext` instead. But the question text itself remains a free-text field, and that field stays adversarial data. The guarantee is therefore **attenuation, not elimination**: the model can still be influenced by that text; it cannot call anything outside `{lookup}`, because the handle refuses. Blast radius drops from arbitrary tool execution to a bad lookup query.

Two things follow that are easy to get wrong in the reading. The confined tier does **not** close this residual — what bounds the damage is the capability scope, not the sandbox — which is why the assertion is made on that tier on purpose, so that stronger enforcement is not misread as a stronger claim. And the attenuation depends on schema discipline, not on the framework alone: a discharging type that retains unbounded free text in a position flowing to a privileged node gives most of it back. The framework supplies the enforcement substrate; disciplined schema design is what makes use of it.

### 4.4 Enforcement tiers: what each one stops

The same escape attempts were run on both tiers. The host-tier column is **expected** to read ESCAPES: host discipline gives a node only its declared handles, but nothing stops a hostile node from reaching around the object model. That gap is the reason the confined tier exists, and it is reported here rather than omitted.

| **escape attempt** | **host tier** | **confined tier** | |
|---|---|---|---|
| read a file | ESCAPES | denied | ✓ |
| open a socket | ESCAPES | denied | ✓ |
| read an env var | ESCAPES | denied | ✓ |
| call an ungranted capability | ESCAPES | denied | ✓ |

Table 4: Escape attempts on both tiers. The host tier's failures are the recorded gap, not an omission.

The final row is the sharpest: the component imports an interface it was never granted, so it cannot instantiate at all — the refusal lands before any guest code runs. And on the confined tier the capability is **absent, not merely unexposed**. An inference-only node's import set holds only `aap:caps/inference-llm@0.1.0`, with 0 filesystem, socket, environment, or clock imports in it. Confinement is a property of the artifact rather than of how the host happened to configure it — the distinction Section 3.3.2 is built to make.

**Fidelity.** Enforcement is unforgeable at the WASM boundary, not at the memory level; CHERI remains a named follow-up (Section 7.3). Only the nodes ported to the confined tier get it — the rest run on the host tier, which demonstrates the shape of confinement rather than enforcing it. That the two tiers compose in one graph is the incremental-migration path, and it is the reason the host tier's gap is a transitional state rather than a defect of the model.

## 5 Predictions and outcomes

The founding vision [1] made a set of claims about a system that did not exist, and hedged them accordingly: properties that *would* hold in a sound realisation, guarantees that *would* be structural. This section states, claim by claim, which of them the demonstrator substantiates, which it substantiates only in part, and which remain exactly as conditional as they were. It is where a reader should look first to know what has actually changed.

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 that the vision did not attach to it. **Conditional** means the claim is unchanged: still hedged, still unproven, and 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** | **Where** |
|---|---|---|
| Unsafe wirings are rejected statically, before execution | Substantiated | Section 4.1 |
| Trust cannot be laundered by relabelling the consumer | Substantiated | Section 4.1 |
| A node cannot exceed its declared capabilities | Partial | Section 4.4 |
| Nodes have no ambient authority | Partial | Section 4.4 |
| Per-crossing overhead stays inside the stated envelope | Substantiated | Section 4.2 |
| Prompt injection is structurally attenuated | Substantiated | Section 4.3 |
| Hierarchical composition follows from the model | Substantiated | Section 3.5 |
| Capability identity, revocation, and rotation are expressible | Substantiated | Section 3.4 |
| Implementations are interchangeable across languages | Partial | Section 3.3.2 |
| The *declared* capability distribution is complete and reviewable | Substantiated | Section 3.2 |
| Well-typed wiring implies noninterference | Conditional | Section 7.3 |
| The full type system (graded or decentralised labels) | Conditional | Section 7.1 |
| User-level authorisation threaded through capability injection | Conditional | Section 7.1 |
| Behavioural contracts constrain generated implementations | Conditional | Section 7.2 |
| Memory-level unforgeability on CHERI hardware | Conditional | Section 7.3 |
| Replay and time-travel debugging from boundary logs | Not attempted | Section 7.2 |
| Differential evaluation of the graph at scale | Not attempted | Section 7 |
| Agent tooling and the visual graph editor | Not attempted | Section 7.1 |

Table 5: The founding vision's claims and what the demonstrator does to each. Every "Substantiated" is qualified by the demonstrator's scale; see Section 8.

## 5.1 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: the dependency-free validator rejects both 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, not 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 the vision's unmeasured assertion that per-crossing cost under roughly 1 ms would keep overhead acceptable, a crossing costs 55.6 µs 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 we measured. Hierarchical composition was where the artifact was more generous than the prediction: the vision's "falls out naturally from the model" reads as optimism, but proved accurate in a checkable way (Section 3.5) — 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 enforce had no mechanism by which it could fail.

## 5.2 What it substantiates only in part

Three claims hold under restrictions the vision did not attach to them, and stating those restrictions is more useful than 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 — now a majority of the demonstrated customer path (5 of its 6 nodes, Section 3.3.2) rather than two isolated nodes, so the claim holds for most of the path rather than a fragment of it. What keeps it **Partial** is unchanged by that coverage: the host tier around those nodes still escapes, and its escapes are recorded as passing tests and reported in Table 4 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: as Section 3.3.2 recounts, "no ambient authority" turns out to name two properties — an earlier WASI-context build was confined only by the host's configuration (its modules still **imported** `fd_write` and `path_open`), whereas the current components import no such function at all — and only the second, a property of the artifact, is worth claiming. Building it twice is what made the distinction visible.

**Implementations are interchangeable across languages** is demonstrated for 5 node bodies now, not one, and beyond the pure-transformation case. Each confined node runs from a Rust body regenerated from the same signature and contract as its Python counterpart with the graph unchanged (Section 3.3.2), and the set spans the typed capability boundary rather than a single inference node: `FetchContext` reads through a knowledge-base handle, `GenerateResponse` calls a scoped tool, and `SendReply` performs an identity-routed send that lands on the same channel instance whichever body runs. It stays **Partial** rather than **Substantiated**, and the restriction is the point: a contract is a partial specification, so two implementations can satisfy it and still differ observably (Section 7.2), and the demonstrator holds its bodies to tests rather than to a contract language it does not yet have. Existence across several nodes and capability kinds — not a claim about the general case, whose meaning that incompleteness still bounds.

## 5.3 What remains conditional

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 [32] is open. The vision's conditional mood for these claims was correct, and this paper keeps it rather than quietly upgrading it to the

present tense used around it — the discipline most easily lost when everything adjacent reports a built result.

User-level authorisation, on which the confused-deputy attenuation argument depends, remains undesigned. Behavioural contracts — the mechanism that is supposed to make generated implementations verifiable and interchangeable — have no language and no checker; the demonstrator has tests, which is not the same thing. CHERI integration is untouched, so enforcement stops at the WASM boundary. Replay, differential evaluation, agent tooling, and the visual editor were not attempted at all, and this paper offers no evidence about them in either direction.

### 5.4 What building it revealed that the vision did not see

The most useful output of an implementation is the part of the design it corrects, and 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 [29] — 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 (Section 3.4). 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.

**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 what admits the laundering graph. This simplified the design rather than complicating it — a case where the artifact argued the vision into a cleaner position.

**Confinement is a property of an artifact or of a configuration, and the vision conflated them.** The distinction is drawn in Section 3.3.2 and its consequence recorded in Section 5.2; it is named again here because it is the clearest instance of a security claim that sounds identical in prose while naming two different guarantees, and only building it twice distinguished them.

**The union-alias convention for sub-graph outputs needed a check the vision did not call for.** The vision treated that convention as the simplest of several designs and moved on; in the demonstrator it proved the cheapest to adopt and, at first, the least self-checking — `ServiceOutcome` was asserted in the JSON and verified by 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: spell the union structurally rather than name an alias the language cannot resolve, and check the declared output against the referenced graph's terminals (Section 3.5). The convention is usable; it is not self-checking, and the lesson generalises — a boundary type that abbreviates must still be verified against what it abbreviates.

## 6 Related work

Each element of this work has been explored independently; the convergence that makes their synthesis newly practical is the argument of Section 1. To our knowledge, no existing system simultaneously provides graph-level capability analysis, trust-propagated type checking across component boundaries, and generated implementations executing within capability-restricted sandboxes; the security-by-construction property emerges from their combination, not from

any element alone. The one substantial addition to the prior art surveyed in the earlier paper is the WASM Component Model's interface layer (Section 6.9), on which the confined tier of Section 3.3.2 is built directly.

## 6.1 Model-driven architecture and its lessons

The Object Management Group's Model-Driven Architecture [4], launched in 2001, pursued a superficially similar vision — models as the primary artifact, code generated from them — and foundered on round-trip engineering: models and generated code diverged as soon as developers edited the code, and re-synchronising them cost more than maintaining the code alone. The signal graph avoids this structurally. It is not a model *of* the code but *is* the program at the level humans review, and node implementations are regenerable artifacts against the stable interface of the graph's signatures and contracts, so there is no return trip to keep in sync.

## 6.2 Architectural modelling: C4

The C4 model [34] is the most widely adopted lightweight architecture-diagramming approach (Structurizr, LikeC4, IcePanel, Mermaid). Recent work extends it toward the role envisaged here: LikeC4 [35] exposes the model to agents as a queryable knowledge base, and some practitioners, including the author [36], already maintain C4 models as "executable context" that constrains agent behaviour. C4′s limit is that it is a *communication* model, not a *constraint* one — it describes architecture without enforcing it. The signal graph is the typed counterpart that both describes and enforces: the diagram and the program are one artifact.

## 6.3 Effect systems and purity: Haskell, Idris, and beyond

Haskell showed that purity-by-default with explicit effects is practical [37]: `a → IO b` declares in the signature that a function performs effects, while `a → b` is guaranteed pure. Algebraic effect systems (Koka [38], Frank [39]) and dependent types (Idris 2 [40]) make effects first-class, parameterisable, and composable. Roc [41] is the closest language-level analogue to per-node capability injection: side effects come only from an interchangeable "platform" the application cannot bypass, so the platform boundary *is* the capability boundary — though at whole-application rather than per-component granularity. A more recent line makes capabilities the type-level currency for effects directly: Effekt [42] passes effect operations as capability values the type system tracks in scope, and Scala 3′s capture checking [43] records in a value's type which capabilities it *captures*. Both show capability tracking can be ergonomic inside a general-purpose type system, at function granularity within one program. This work applies the same principle — effects declared in signatures, not drawn from ambient context — one level up, at architectural components, with the runtime rather than the compiler as enforcer; showing that this coarser enforcement suffices for the security properties claimed is part of the verification obligation of Section 7.

## 6.4 Content-addressed code: Unison

Unison [44] stores code as a database of hash-identified typed ASTs rather than text files, so the codebase is always type-checked, compilation is perfectly incremental, and whole classes of merge conflict disappear; its "abilities" are an algebraic effect system in which a program can perform only effects it has been given, and its 1.0 (2025) shows the model is production-viable. Darklang [45] pursued a unified code-editor-infrastructure environment and showed real productivity gains for simple services, but its proprietary, hard-to-scale substrate is a cautionary tale: adoption needs an incremental migration path, which is why this work layers the signal graph above existing runtimes (WASM, BEAM) rather than replacing them. Nix [46] and Guix make the same content-addressed, pure-input-to-output model work at scale for software builds

— a large existence proof in an adjacent domain. Unison's model is complementary to ours: it addresses code storage and *what category* of effect a function performs, while the signal graph addresses explicit capability wiring and the propagation of `Untrusted<T>` — the fine-grained authority boundaries (a handle scoped to one database versus ambient access) that abilities do not track.

## 6.5 Live programming with typed holes: Hazel

Hazel [47], [48] keeps every editor state statically and dynamically meaningful through *typed holes* that carry type information and propagate as opaque values, so feedback is continuous even on incomplete programs. Two things bear on this work. Its hole calculus gives well-defined types to partial programs — a semantic foundation for the "project downstream effects" step of the Section 2.4 workflow, where an agent's proposed graph is momentarily incomplete. And a 2024 paper from the group [9] integrates LLM generation into the typed-hole environment and finds that static context from the hole's type materially improves generation quality — direct empirical support for the claim that node-implementation agents benefit from the graph's typed signatures.

## 6.6 Process isolation and message passing: Erlang/BEAM

The BEAM virtual machine [49] is the closest existing model to the intended runtime: lightweight, fully isolated (no shared memory) processes communicating only by message passing, from which "let it crash" resilience follows. The signal graph's isolated typed-message components are an instance of the actor model [50], but the actor model as usually realised (Erlang, Akka, Orleans) leaves two things implicit that the graph makes explicit: which *external resources* an actor may reach (its received-message set is typed, its authority is not), and the wiring, which it determines at runtime by message sends rather than statically. Pony [51] is the closest capability-typed prior art, integrating reference capabilities into the actor type system and carrying them through composition — but its capabilities govern *memory* (aliasing, mutability), whereas the signal graph's govern *external authority*; this work lifts that insight from the memory-reference level to the architectural one and pairs it with trust propagation and an explicit wiring graph.

## 6.7 Capability-based security

The object-capability model [29], [52] makes possession of an unforgeable reference the proof of authorisation, and has been realised across the stack: languages (E [29], Caja [53]), operating systems (Capsicum [54], seL4 [55]), and runtimes (Deno [56], WASI). Most relevant is the *distributed* treatment, where network reachability is itself ambient authority: E mediates all inter-object communication through explicitly passed references with no ambient network [29], Agoric's Hardened JavaScript [57] brings the discipline to a mainstream language, and Stiegler [58] gives an accessible account. The signal graph's explicit edge wiring is the architectural analogue of E's reference passing — a node not wired to a network capability has no mechanism for external communication — which the confined tier realises and the host tier does not (Section 4.4). Cedar [59], a formally verified authorisation language, shows fine-grained analysable authority is production-viable and is philosophically aligned with treating capability wiring as a typed, reviewable artifact. The combination of object-capability security with FRP's signal graph is, to our knowledge, novel as a whole-system substrate — existing capability systems enforce authority at runtime, existing FRP systems enforce dataflow at the type level, and the synthesis does both; Section 4 reports how far the demonstrator carries it.

## 6.8 CHERI: capabilities in hardware

CHERI [60] makes every pointer a hardware-checked capability carrying bounds, permissions, and a tag, so forgery (via overflow, type confusion, or integer-to-pointer cast) traps in hardware with no software guard. It is reaching commercial maturity: Arm's Morello [61] on AArch64, Microsoft's CHERIoT [62] on embedded RISC-V, Codasip's commercial IP [63], and SCI Semiconductor's ICENI [64], which reached first silicon in March 2026 [65] — described as the first commercial CHERI in silicon; the CHERI Alliance [66] (Google a founding member) coordinates adoption. Three properties matter here: unforgeable capabilities, fine-grained in-process compartmentalisation, and low porting cost — a 2021 study ported six million lines of C/C++ with changes to 0.026% of source lines [67], and for code generated against a CHERI-aware runtime the cost would be zero. No part of this work has run on CHERI hardware (Section 7.3).

## 6.9 Lightweight sandboxing and the WASM Component Model

WebAssembly provides capability-based isolation with near-native performance across languages [68], and two layers on top of it matter here — the durable one being the substrate this work depends on. The *WASM Component Model* [69] is the interface-type layer: components declare typed imports and exports (in the WIT interface language) that the runtime links with type-checked bindings. The *WebAssembly System Interface* (WASI) is a standard *library* of such interfaces — filesystems, clocks, sockets — built on the Component Model. The distinction is what the confined tier exploits: it expresses each capability kind as its own WIT interface, so a node's `with` clause *is* its component's import list and the runtime boundary realises the node's typed signature rather than sitting beneath it as a byte protocol. WASI is itself a worked precedent for that pattern — its I/O model grants each resource (`wasi:filesystem`, `wasi:clocks`, `wasi:sockets`) as an explicit typed interface, structurally the same object as `DBHandle<'knowledge-base', read>` applied to host resources — so the signal graph does not adopt WASI so much as recognise its interfaces as one instance of capability-as-interface and extend the discipline to the application capabilities WASI does not model.

The layering also settles a longevity worry, since WASI's interfaces are still moving: 0.1′s `wasm32-wasip1` snapshot is legacy, 0.2 rebased onto the Component Model, 0.3 [33] added native async and folded `wasi:io` into the canonical ABI, and a stable 1.0 is expected [70]. Every version rebases onto the same Component Model — the part advancing toward standardisation — so betting on the substrate rather than any WASI snapshot, as the confined tier does by defining its own WIT interfaces (Section 3.3.2), leaves it untouched by that churn; where a node genuinely needs a host resource, the corresponding WASI interface is the right thing to grant it, as a capability like any other. Beyond WASM, Monty [71] reaches microsecond-scale startup with host isolation by default, BEAM processes start in single-digit microseconds, and CHERIoT demonstrates hardware compartmentalisation with negligible overhead. CHERI would also backstop a software sandbox — preventing capability forgery at the memory level even under a sandbox memory-safety bug — for two complementary enforcement layers once integrated (Section 7.3), a combination this work does not yet have.

## 6.10 Durable execution and replay

Temporal [72], Restate [73], and Azure Durable Functions [74] provide production-grade state persistence and deterministic replay from event logs, validating the mechanism behind Section 2.1.3 at scale. Their persistent difficulty is nondeterministic interleaving — replay must reproduce concurrent ordering — which they manage by constraining application code (Temporal) or journalling each operation's outcome (Restate). The signal graph's pure,

topology-ordered propagation would avoid most of this at the inter-node level, with one residual (the fan-in merge order) discussed under Section 7; replay for nodes with internal concurrency remains open. Where the model *would* differ is granularity and integration: replay at *capability-boundary crossings* rather than workflow steps, with the type system's trust and capability annotations carried into the replay infrastructure. These are would-claims — replay is Section 7.2 work, not attempted here.

### 6.11 Spec-driven development

SDD frameworks — OpenSpec [10], Spec Kit [11], Kiro [12] — persist versioned specification artifacts so agents have stable context instead of chat history. Codespeak [75] goes further, compiling plain-English specs to code and treating implementation as generated — close to this work's stance, but its specs are untyped prose without capability or trust annotations, so the security-by-construction properties are outside its scope. Current frameworks treat spec and architecture as separate prose concerns; this work's argument is that as they mature their output should *become* the typed graph — a proposed change is a graph transformation, and the spec/architecture distinction dissolves when the graph is both. A parallel, less formal current supports the trajectory: visual automation platforms (Zapier [76], Make.com [77], n8n [78]) already let non-developers build directed graphs of actions, n8n's AI builder runs exactly the intent → generated-graph → visual-review loop at a higher abstraction, and agent-orchestration frameworks such as LangGraph [79] structure LLM applications as explicit state-transition graphs. What these lack is what this work adds — typed interfaces, capability restriction, trust propagation — and as teams scale from internal automations to customer-facing agents, that gap becomes the acute one.

## 7 Research 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, organised in three phases of increasing scope and 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. Each item is an open problem, and where the demonstrator has narrowed one, the narrowing is stated so the remaining question is not overclaimed as smaller than it is.

### 7.1 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 [16] and algebraic effect systems [38] 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 (Section 3.2). What remains open is the choice among two-point, graded (`Untrusted ⊑ Sanitised ⊑ Trusted`), and Jif-style decentralised-label [32] designs. The condition any successor must meet can be stated concretely, and is worth stating because it 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 [30, §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 concrete graph demonstrates 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. 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 now settled **at the runtime** (Section 3.5), 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 (Section 3.4) 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 than the input side in its **design**, though the demonstrator has since 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 `ServiceOutcome = DeliveryConfirmation | EscalationTicket` and 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), now with the output side **checked**: the cross-graph analysis requires a sub-graph node's declared boundary type to equal the union of the referenced graph's terminal outputs (Section 3.5), so the convention is at least verified against what it abbreviates rather than merely asserted. What stays open is the design, not the check — whether (ii) or (iii) is preferable to the union convention — 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. These are design questions, not implementation details.

**User-level authorisation threaded through capability injection.** 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. Whether capabilities carry an authenticated principal, whether they compose under delegation, and how sub-graph boundaries preserve the principal across composition are all undesigned. This matters because the graph-level security argument leans on capabilities being both scoped to a resource **and** bound to an authorised principal; a weak binding would undermine the confused-deputy and privilege-escalation claims of Section 2.6.

**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. This is inseparable from **graph-scale comprehension**, which the demonstrator does nothing to address — the concrete example has nine nodes and real systems have hundreds. 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.

## 7.2 Phase 2: hardening and deployment

**Shallow verification.** Develop tooling to confirm that generated node implementations satisfy their declared contracts — which first requires a contract language, of which the demonstrator has none. 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* [80] 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.

**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 two-tier composition of Section 3.3 is the demonstrator's 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 the enforcement of Section 4.4 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 (Section 3.4). 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.

## 7.3 Phase 3: formal foundations and hardware

**Type-system soundness.** The security properties of Section 2.6 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 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 [81] 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 of Section 2.6 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 [62] 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 [29] and Agoric's Hardened JavaScript 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.

# 8 Threats to validity

The demonstrator is small and the claim in Section 1.1 was scoped to what it can carry. This section states the limits explicitly, because the failure mode of a paper like this one is not a false claim but a true claim read more broadly than it was made.

## 8.1 Construct validity: what the corpus measures

**The corpus is curated, and counts are not coverage.** Table 1 reports that 3 of 3 unsafe wirings were rejected. That ratio is 100% by construction: the corpus contains the mistakes we thought to write down, and we wrote down mistakes we expected the analyses to catch. A fully-caught curated corpus reads as a soundness proof unless it says otherwise, and it is not one. The honest reading is narrow — the analyses are implementable and catch the classes they target — and no claim is made that an uncaught class does not exist.

**Reason-class pinning mitigates but does not resolve this.** Requiring `launder_trust` to be caught by the trust lattice rather than by edge typing means the corpus cannot stay green while the mechanism under test quietly stops working. That is a real guard against rot, and it is why the pins exist. It says nothing about mistakes absent from the corpus.

**Two graphs, one domain.** Both canonical graphs describe an AI customer-support pipeline, chosen because untrusted input, graduated LLM access, and fine-grained capability distinctions

are all visible in it. It is a domain selected to display the properties under test. Whether the analyses are as informative on graphs whose security structure is less legible is untested.

## 8.2 Internal validity: the measurements

**The overhead figures are from one machine and a two-node slice.** They are wall-clock timings on `Linux-6.17.0-1020-azure-x86_64-with-glibc2.39`, reported to establish an order of magnitude, not as portable benchmarks. The supported claim is that a crossing costs tens of microseconds and therefore sits well inside the asserted envelope — not the specific figure.

**The per-crossing cost is a differenced quantity, which is fragile.** It is derived by differencing two warm timings through one component, so any systematic cost landing in one term and not the other is charged directly to the boundary. That this is not hypothetical is the subject of Section 4.2: an earlier benchmark timed a single cold pass, which was enough to distort the figure by an order of magnitude.

**Instantiation and compilation costs are reported but not amortised.** The per-node instantiation cost is the tier's fixed price today; instance pooling would reduce it, and has not been implemented. Serialisation cost for complex types at node boundaries is not measured at all.

## 8.3 External validity: scale and generalisation

**Nine nodes, not hundreds.** The concrete graph has nine nodes and the composition graph three services. Every claim about reviewability, comprehension, and the graph being "visible at a glance" is made at that scale, and graph-scale comprehension is an unaddressed Phase 1 obligation (Section 7.1). The visual programming and model-driven traditions supply ample evidence that this is where such systems become difficult, and the demonstrator offers no evidence that this one does not.

**Two nodes on the confined tier, not all of them.** The enforcement results of Section 4.4 apply to nodes compiled to components. The rest run on the host tier, where the same escape attempts **succeed**. A reader who takes "the demonstrator confines nodes" as a property of the system rather than of the ported nodes has the wrong picture — the two-tier arrangement is a migration path, and the host tier's gap is its cost.

**A Python host and a proof-of-concept runtime.** There is no differential evaluation, no snapshotting, no production deployment, and no concurrent execution. The runtime propagates signals along an active path in a single thread.

## 8.4 Threats to the security argument itself

These are the limits most consequential for how the paper should be read.

**No soundness result.** Nothing here connects well-typed wiring to noninterference. The validator implements a lattice; it does not prove that graphs well-typed under that lattice have the property the lattice is supposed to deliver. Every security claim in this paper is therefore a claim about **what the implemented checks reject**, not about what a sound type system would guarantee.

**Enforcement stops at the WASM boundary.** Confinement on the confined tier is unforgeable at that boundary and not at the memory level. A sandbox escape via a `wasmtime` defect is out of scope, and CHERI remains a named follow-up rather than an implemented layer.

**The free-text residual is real and is not closed by anything here.** Table 3 reports that adversarial text survives in a permitted field on the confined tier. That row is asserted

deliberately so that stronger enforcement is not misread as a stronger claim. Prompt injection is attenuated — the blast radius drops from arbitrary tool execution to a bad lookup query — and what bounds it is the capability scope, not the type and not the sandbox.

**Covert channels and distributed authority are out of scope (Section 7.2, Section 7.3).** A node with a permitted capability can encode information in the timing, shape, or ordering of its legitimate outputs; a node with a network capability can in principle acquire authority out-of-band that the graph does not model. Neither is addressed.

### 8.5 Threats from the method

The demonstrator was implemented by an AI agent against author-directed specifications, and this paper was 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 the reason the 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 figures in Section 4 are interpolated from a single run rather than transcribed, and why the properties this paper asserts are, wherever practical, backed by tests rather than by sentences in the paper.

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

## 9 Conclusion

The founding vision [1] argued that AI coding agents had removed the historical obstacle to graph-based code representations, and that the opportunity this creates is not merely a better diagram but a substrate in which the architecture model, the security policy, and the program are one artifact. It made that argument with nothing running behind it, and hedged accordingly.

This paper reports the demonstrator built to test it. Unsafe wirings are rejected at assembly time, before any node runs, each by the analysis meant to catch it — including the laundering case that type-checks on every edge and is caught instead as a lattice violation. Node bodies compiled to WebAssembly components import exactly the capability interfaces their signatures declare and nothing else, so confinement is a property of the artifact rather than of the host's configuration; escape attempts that succeed on the host tier are refused there. A capability-boundary crossing costs 55.6 µs, inside the envelope the vision asserted without evidence, while marshalling typed values rather than opaque bytes. Hierarchical composition executes, and a sub-graph cannot mint authority of its own because there is no mechanism by which it could.

The distance remaining is larger than the distance covered, and Section 5 and Section 8 are the parts of this paper we would most want read. No soundness argument connects well-typed wiring to noninterference; the corpus is curated, so its counts are evidence of implementability rather than of coverage; the host tier's escapes are a real gap that the two-tier arrangement makes transitional rather than acceptable; and prompt injection is attenuated, not eliminated — adversarial text still reaches the tool-capable node in a permitted field, and what bounds the damage is the capability scope. The contract language, the user-level authorisation binding, replay, the agent tooling, and CHERI enforcement are all untouched.

What the exercise most changed is the design rather than the confidence. Building it showed that trust checking and edge typing are one order rather than two, that revocation is not well-posed until capability **identity** is expressible in the graph source, that "no ambient authority" names two different guarantees of which only one is worth claiming, and that the cheapest

convention for sub-graph outputs is the one hardest to verify. None of those corrections were visible from the vision's vantage point; all of them came from the artifact arguing back. That is the case for stating a design's predictions before building it and then reporting, without revision, what held — as much as it is a case for the signal graph itself.

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

[2] JetBrains, "MPS: The Domain-Specific Language Creator." [Online]. Available: https://www.jetbrains.com/mps/

[3] C. Simonyi, M. Christerson, and S. Clifford, "Intentional Software," in *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, ACM, 2006, pp. 451–464. doi: 10.1145/1167473.1167511.

[4] Object Management Group, "MDA Guide revision 2.0," technical report ormsc/14–6–1, 2014. [Online]. Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01

[5] Anthropic, "Claude Code: An Agentic Coding Tool." [Online]. Available: https://docs.anthropic.com/en/docs/claude-code

[6] OpenAI, "Codex CLI: Lightweight Coding Agent." [Online]. Available: https://github.com/openai/codex

[7] Anysphere, "Cursor: The AI Code Editor." [Online]. Available: https://cursor.com/

[8] Codeium, "Windsurf: The Agentic IDE." [Online]. Available: https://windsurf.com/

[9] A. Blinn, X. Li, J. H. Kim, and C. Omar, "Statically Contextualizing Large Language Models with Typed Holes," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 468–498, Oct. 2024, doi: 10.1145/3689728.

[10] Fission AI, "OpenSpec: Spec-Driven Development for AI Coding Assistants." [Online]. Available: https://github.com/Fission-AI/OpenSpec

[11] GitHub, "Spec Kit: Toolkit for Spec-Driven Development." [Online]. Available: https://github.com/github/spec-kit

[12] Amazon Web Services, "Kiro: Agentic AI Development from Prototype to Production." [Online]. Available: https://kiro.dev/

[13] C. Elliott and P. Hudak, "Functional reactive animation," in *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, Amsterdam The Netherlands: ACM, Aug. 1997, pp. 263–273. doi: 10.1145/258948.258973.

[14] Z. Wan and P. Hudak, "Functional reactive programming from first principles," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, Vancouver British Columbia Canada: ACM, May 2000, pp. 242–252. doi: 10.1145/349299.349331.

[15] J. Hughes, "Generalising monads to arrows," *Science of Computer Programming*, vol. 37, no. 1–3, pp. 67–111, May 2000, doi: 10.1016/S0167-6423(99)00023-4.

[16] 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.

[17] E. Czaplicki, "Elm: Concurrent FRP for Functional GUIs," Senior thesis, 2012. [Online]. Available: https://elm-lang.org/assets/papers/concurrent-frp.pdf

[18] RxJS Contributors, "RxJS: Reactive Extensions Library for JavaScript." [Online]. Available: https://rxjs.dev/

[19] A. Staltz, "Cycle.js: A Functional and Reactive JavaScript Framework." [Online]. Available: https://cycle.js.org/

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991, doi: 10.1109/5.97300.

[21] G. Berry and G. Gonthier, "The Esterel synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[22] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: the SIGNAL language and its semantics," *Science of Computer Programming*, vol. 16, no. 2, pp. 103–149, 1991.

[23] J. P. Morrison, *Flow-Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, 1994.

[24] National Instruments, "LabVIEW: Systems Engineering Software." [Online]. Available: https://www.ni.com/en/shop/labview.html

[25] MathWorks, "Simulink: Simulation and Model-Based Design." [Online]. Available: https://www.mathworks.com/products/simulink.html

[26] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential Dataflow," in *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR 2013)*, 2013. [Online]. Available: https://www.semanticscholar.org/paper/Differential-Dataflow-McSherry-Murray/f5df61effe8047eb9ea1702cfcc268dbba678567

[27] Materialize, "Materialize: Operational Data Warehouse." [Online]. Available: https://materialize.com/

[28] M. Budiu, T. Chajed, F. McSherry, L. Ryzhyk, and V. Tannen, "DBSP: Automatic Incremental View Maintenance for Rich Query Languages," *Proceedings of the VLDB Endowment*, vol. 16, no. 7, pp. 1601–1614, Mar. 2023, doi: 10.14778/3587136.3587137.

[29] M. S. Miller, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control," PhD Thesis, 2006. [Online]. Available: https://www.semanticscholar.org/paper/Robust-composition%3A-towards-a-unified-approach-to-Shapiro-Miller/f59585e6405581fe7d4cad04648d6bd8a587901a

[30] 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.

[31] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible Dynamic Information Flow Control in Haskell," in *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*, ACM, 2011, pp. 95–106. doi: 10.1145/2034675.2034688.

[32] 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.

[33] WebAssembly WASI Subgroup, "WASI 0.3.0." [Online]. Available: https://github.com/WebAssembly/WASI/releases/tag/v0.3.0

[34] S. Brown, "The C4 Model for Visualising Software Architecture." [Online]. Available: https://c4model.com/

[35] LikeC4, "LikeC4: Architecture as Code with C4 Model and MCP Server." [Online]. Available: https://likec4.dev/

[36] Y. Lavi, "C4 Architecture Skill: AI-Agent-Friendly Architecture Discovery for Claude Code." [Online]. Available: https://github.com/CodelianceAI/Codeliance-Skills/tree/main/plugins/c4-architecture

[37] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. [Online]. Available: https://www.haskell.org/onlinereport/

[38] 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.

[39] S. Lindley, C. McBride, and C. McLaughlin, "Do be do be do," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, Paris France: ACM, Jan. 2017, pp. 500–514. doi: 10.1145/3009837.3009897.

[40] E. Brady, "Idris 2: Quantitative Type Theory in Practice," *LIPIcs, Volume 194, ECOOP 2021*, vol. 194, p. 9:1–9:26, 2021, doi: 10.4230/LIPICS.ECOOP.2021.9.

[41] R. Feldman, "Roc: A Fast, Friendly, Functional Language." [Online]. Available: https://www.roc-lang.org/

[42] J. I. Brachthäuser, P. Schuster, and K. Ostermann, "Effects as capabilities: effect handlers and lightweight effect polymorphism," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, p. 126:1–126:30, Nov. 2020, doi: 10.1145/3428194.

[43] A. Boruch-Gruszecki, M. Odersky, E. Lee, O. Lhoták, and J. Brachthäuser, "Capturing Types," *ACM Transactions on Programming Languages and Systems*, vol. 45, no. 4, Nov. 2023, doi: 10.1145/3618003.

[44] P. Chiusano, R. Bjärnason, and A. Irani, "Unison: A Content-Addressed Functional Programming Language." [Online]. Available: https://www.unison-lang.org/

[45] P. Biggar and E. Schuster, "Dark: A Holistic Programming Language, Editor, and Infrastructure." [Online]. Available: https://darklang.com/

[46] E. Dolstra, "The Purely Functional Software Deployment Model," PhD Thesis, 2006. [Online]. Available: https://edolstra.github.io/pubs/phd-thesis.pdf

[47] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer, "Hazelnut: a bidirectionally typed structure editor calculus," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, Paris France: ACM, Jan. 2017, pp. 86–99. doi: 10.1145/3009837.3009900.

[48] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer, "Live functional programming with typed holes," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–32, Jan. 2019, doi: 10.1145/3290327.

[49] J. Armstrong, "Making Reliable Distributed Systems in the Presence of Software Errors," PhD Thesis, 2003. [Online]. Available: https://www.semanticscholar.org/paper/Making-reliable-distributed-systems-in-the-presence-Armstrong/2048676806baee4c27934153ae7aa22f17094cec

[50] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*, 1973, pp. 235–245.

[51] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil, "Deny Capabilities for Safe, Fast Actors," in *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2015)*, ACM, 2015, pp. 1–12. doi: 10.1145/2824815.2824816.

[52] M. S. Miller and J. S. Shapiro, "Paradigm Regained: Abstraction Mechanisms for Access Control," in *Advances in Computing Science – ASIAN 2003: Programming Languages*

*and Distributed Computation*, vol. 2896, G. Goos, J. Hartmanis, J. Van Leeuwen, and V. A. Saraswat, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 224–242. doi: 10.1007/978-3-540-40965-6_15.

[53] Google, "Google Caja: A Source-to-Source Translator for Securing JavaScript-based Web Content." [Online]. Available: https://developers.google.com/caja

[54] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical Capabilities for UNIX," in *Proceedings of the 19th USENIX Security Symposium (USENIX Security 2010)*, 2010. [Online]. Available: https://www.usenix.org/conference/usenixsecurity10/capsicum-practical-capabilities-unix

[55] G. Klein *et al.*, "seL4: Formal Verification of an Operating-System Kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, Big Sky Montana USA: ACM, Oct. 2009, pp. 207–220. doi: 10.1145/1629575.1629596.

[56] Deno Land, "Deno: The Open-Source JavaScript Runtime for the Modern Web." [Online]. Available: https://deno.com/

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

[58] M. Stiegler, "An Introduction to E and the Distributed Object-Capability Model." [Online]. Available: http://www.skyhunter.com/marcs/ewalnut.html

[59] 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.

[60] R. N. Watson *et al.*, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, San Jose, CA: IEEE, May 2015, pp. 20–37. doi: 10.1109/SP.2015.9.

[61] Arm, "Morello: An Arm Architecture for Capability-Based Memory Safety." [Online]. Available: https://www.arm.com/architecture/cpu/morello

[62] 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.

[63] Codasip, "Codasip Studio CHERI: Hardware Memory Safety Solutions." [Online]. Available: https://codasip.com/cheri/

[64] SCI Semiconductor, "ICENI: CHERI-Enabled Embedded Processor Family." [Online]. Available: https://www.scisemi.com/products/iceni-device-family/

[65] SCI Semiconductor, "SCI Semiconductor Announces First Silicon of Cybersecure MCU, ICENI." [Online]. Available: https://www.scisemi.com/company/press-release-iceni-first-commercial-cheri/

[66] CHERI Alliance, "CHERI Alliance." [Online]. Available: https://cheri-alliance.org/

[67] R. N. M. Watson, B. Laurie, and A. Richardson, "Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem," technical report, Sep. 2021. [Online]. Available: https://www.capabilitieslimited.co.uk/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf

[68] A. Haas *et al.*, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Barcelona Spain: ACM, Jun. 2017, pp. 185–200. doi: 10.1145/3062341.3062363.

[69] W3C WebAssembly Community Group, "WebAssembly Component Model: Typed Inter-Component Interfaces." [Online]. Available: https://github.com/WebAssembly/component-model

[70] WebAssembly WASI Subgroup, "WASI Roadmap." [Online]. Available: https://wasi.dev/roadmap

[71] Pydantic, "Monty: A Minimal, Secure Python Interpreter Written in Rust for Use by AI." [Online]. Available: https://github.com/pydantic/monty

[72] Temporal Technologies, "Temporal: Durable Execution Platform." [Online]. Available: https://temporal.io/

[73] Restate, "Restate: Durable Execution for Distributed Applications." [Online]. Available: https://restate.dev/

[74] Microsoft, "Azure Durable Functions." [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview

[75] A. Breslav, "Codespeak: A Programming Language Powered by LLMs." [Online]. Available: https://codespeak.dev/

[76] Zapier, "Zapier: Automate AI Workflows, Agents, and Apps." [Online]. Available: https://zapier.com/

[77] Make, "Make: Visual Workflow Automation Platform." [Online]. Available: https://www.make.com/

[78] n8n, "n8n: Fair-Code Workflow Automation Platform with Native AI Capabilities." [Online]. Available: https://n8n.io/

[79] LangChain, "LangGraph: Low-Level Orchestration Framework for Building Stateful Agents." [Online]. Available: https://www.langchain.com/langgraph

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

[81] 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.

[82] 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.