Reaktor Graph: Typed Port Hypergraphs, Traversals and Patches for Multi-Runtime Systems

The design of Reaktor Graph and the Reaktor Graph Language (RGL)

Shibasis
BestBuds Ventures · Reaktor · shibasis0801@gmail.com

8 October 2026 · Design paper

Abstract

An application today is a fleet of cooperating runtimes: phones, browsers, edge isolates, servers, databases, brokers and model hosts. Its contracts, wiring, placement, data, authority and evidence live in separate tools, and the people and agents who change it reconstruct the system from text. We present Reaktor Graph (RG), a typed, multilayer, heterogeneous port hypergraph that serves as the single semantic substrate for such systems, and the Reaktor Graph Language (RGL), its embedding in Kotlin common code. RGL is designed to run natively on every host and, compiled to JavaScript, on an embedded Hermes engine, so that a self-contained subgraph can be shipped as data. RG rests on three primitives: a *structure* (nodes, owned ports, role-labelled hyperedges and a containment forest, typed by an open signature and fibred over six layers), a *computation* over the structure (typed message passing along incidences) and a *change* to it (double-pushout patches). We contribute (i) a minimal signature of fourteen base node kinds built from facets, ports factored into family, role, contract, ownership and quality of service, four connection and ten assertion families, and information-flow labels, tenancy and an effect lattice as facets and rules; (ii) a compositional semantics in which mounting a subgraph is composition in a hypergraph category and replacing one is sound under trace refinement; (iii) theorems on determinism, incremental maintenance and coordination-freedom of traversals, which make dependency injection, navigation, rendering, lifecycle, capability resolution and code-push closure instances of one primitive; (iv) transactional evolution in which independent patches commute and graph diffs replace textual diffs as the unit of review; (v) a data-oriented kernel and a wire protocol, in which reads are traversals, watches are incremental and authorities federate through signed views, with a reference realization in Kotlin on Cloudflare, Google Cloud and Postgres, the capability profile that any other provider must meet, and four further stacks: the reference stack at large scale, a Chinese cloud, another public cloud and an internally hosted corporate stack; and (vi) an integration with the Kotlin compiler in which the graph supplies the facts that thirteen optimisations need, with the conditions under which each is sound. Applied to the 140 published Reaktor design graphs, the type system maps all 648 relations onto its families, reduces 123 role names to 31, finds the family of a connection determined by its port profile in 95% of the connections whose ports imply one, and exposes 21 bindings that silently cross hosts. A survey of 283 features of Reaktor and five systems built on it needed no new kind or family. Five TLA$^+$ templates, checked with TLC, show why fencing at the store, stable effect identities, transactional outboxes, drained releases and gap-free watches are part of the design. We take Unreal Engine's Blueprints as the gold standard for typed graphs over a native core, and map twenty recurring problems and eleven classes of multi-runtime systems onto the model, including scientific simulation and neurosymbolic AI. We describe how compilers, test and analysis tools, agents and teams of people and agents work on one graph, argue that a typed system graph is how programming keeps pace as software outgrows the teams that run it, and close with fifteen hypotheses and a twelve-month plan whose acceptance criteria can reject the design.

Keywords: port hypergraphs; attributed C-sets; hypergraph categories; double-pushout rewriting; message passing; multitier and choreographic programming; information-flow control; dynamic software update; TLA$^+$; AI agents; edge computing; Kotlin Multiplatform.

Status of claims. Reaktor's current graph runtime runs in production in a social application and its edge workers; every statement about it in Section 2 cites its source. Reaktor Graph and RGL are specified here and are not yet implemented. The exceptions are the hypergraph definition IR that the current runtime already contains, which RG adopts as its seed, and the artefacts in Section 20, which were executed for this paper and are included with it. The reference realization (Section 13) is the stack we build on first; it is a choice, not a requirement of the design. Listings show proposed interfaces. Theorems are proved in Appendix A or cited. Statements about precedents describe the cited systems, not Reaktor.

Contents

1 Introduction 2
2 The current runtime and what it cannot express 4
3 Preliminaries 6
4 Precedents: what game engines already know 7
5 The Reaktor Graph 13
6 The type system 15
7 Composition and behaviour 22
8 Traversals: one computational primitive 23
9 Change: patches, transactions, diffs and code push 26

10 RGL: the Reaktor Graph Language 29
11 The kernel: a data-oriented realization 36
12 RGP: the Reaktor Graph Protocol 38
13 A reference realization 45
14 Reaktor: modules and tools around the graph 53
15 Assurance inside the graph 57
16 Problems of multi-runtime systems 59
17 Systems in scope 62
18 Programming at the scale of systems 71
19 Tools, agents and teams on one graph 73
20 Evaluation 76
21 Related work 79
22 Implementation plan 81
23 Limitations and open problems 82
24 Conclusion 83
A Proofs 95
B From the 2026 catalogue to Σ_{std} 96
C The bounded models and the corpus script 98
D Study plan 104
E Primers 106
F Glossary 109
G Roadmaps for learning and building 110

1 Introduction

Consider a social application in production on Reaktor's current runtime. It is not one program. Its screens run on Android and iOS; its web client runs in browsers; its API runs as Kotlin/JS on Cloudflare Workers and Durable Objects; graph and batch work runs on JVM services; and its state lives in Postgres, an object store and a graph database, joined by Pub/Sub, queues and WebSockets. Each runtime has its own deployment unit, scheduler, failure behaviour and debugger. The *system* (which screen invokes which operation, through which binding, under which authority, against which data, placed where, observed how) exists in no single place. People reconstruct it from code and dashboards, and AI agents increasingly reconstruct it from text. When an agent proposes a change, it emits a textual diff whose effect on that system has to be reconstructed once more. The gap widens every year: agents multiply the code each person can produce, and each system spans more runtimes, while the attention of the people responsible for it does not grow. We argue in Section 18 that keeping pace requires raising the unit of programming from text to the typed system, and state the claim as a falsifiable hypothesis.

Kubernetes showed that a typed object store with reconciling controllers can act as an operating system over a fleet of machines [47, 347]. Its objects are infrastructure, its relations are labels and selectors, and its semantics stop at the container boundary. We ask what the corresponding substrate is for applications that span many runtimes. Its objects are screens, routes, operations, topics, workflows, tables, places, activations and claims. Its relations are typed and often multi-party. Its controllers include the deployer, and also the dependency injector, the router, the renderer, the lifecycle manager, the code-push planner and the verifier.

Thesis. *A multi-runtime system is a typed, layered port hypergraph. Every platform service is a traversal of that graph or a patch to it, and the behaviour of the system is the composition, along the graph, of the behaviours of its parts.*

Six systems shaped this thesis. We state what each contributes and what we decline to copy.

- **React** [242] expresses all of UI with one primitive and keyed reconciliation. RG makes reconciliation the meaning of its runtime layer: the tree of activations is the reconciled image of the definitions, keyed by stable identity (Section 8.4).
- **Actors** [7, 20, 169] give isolation, mailboxes, supervision and, in Orleans, virtual activation [34]. RG keeps each of these as a facet or a hyperedge family, not as a second runtime (Section 6).
- **Unreal Blueprints** [110] expose durable C++ primitives to visual composition through typed pins and separate execution pins from data pins. Shipped in 2014, before language models could write code, they showed that typed graphs over a native core let mixed teams build software that is both complex and fast, and we treat them as the gold standard (Section 4). RG separates the Transition family from Bind and Channel in the same way, and compiles graph-owned behaviour regions to Kotlin.
- **Transformers** [344] replaced bespoke architectures with one general, parallel primitive: attention, a form of message passing over a set [346]. RG makes the analogous bet for platform services. Typed message passing over incidences is the one computational primitive the kernel provides over the graph, and solvers are called from it (Section 8). Agents act on systems through typed queries and patches rather than text.
- **TLA$^+$** [212, 214] describes systems as state machines, composition as conjunction and implementation as refinement. RG derives specifications from hyperedge families and stores claims and evidence as graph elements (Sections 7 and 15).

- **Kubernetes** contributes the open, typed schema (custom resources) and controllers that compute plans from desired and observed state. RG has an open signature and executors that apply plans computed by pure traversals.

From the current runtime to RG. Reaktor's current graph runtime already composes a production application: nested graphs, typed provider and consumer ports, routes with typed payloads, actors, service clients and Compose rendering (Section 2). It is, however, a binary port graph of mutable objects. It has one relation type in code, random identities, lifecycle and coroutine scopes fixed inside classes, a hard dependency on platform libraries, and no model of change. Publish/subscribe, queues, workflows, FFI, data, capabilities and evidence sit outside it or are attached to it by convention. The 2026 design corpus contains a hypergraph definition IR and a catalogue of 40 node kinds, 24 port profiles and 36 hyperedge kinds [291, 292]. Nothing executes that IR. Its role vocabularies conflict, and the catalogue has no formal semantics. This paper turns those ingredients into one model. Reaktor, as released, consists of RG and RGL, the modules and tools that are fragments of its signature (Section 14), and a reference realization in Kotlin on Cloudflare, Google Cloud and Postgres (Section 13).

Contributions.

C1 A model (Section 5). RG is a typed port hypergraph with a containment forest, formalised as attributed C-sets [280] typed over a signature and fibred over six layers: source, definition, deployment, runtime, observation and assurance. Layer projections are restriction functors, so every restriction view commutes with every patch.

C2 A minimal type system (Section 6). Fourteen base node kinds composed from facets; ports typed by family, role, contract, ownership and quality of service; four connection families (Bind, Session, Channel, Transition) and ten assertion families. Protocol is separated from realization, which is why "rpc" and "ffi" are bindings rather than port kinds. Information-flow labels, tenancy and an effect lattice are facets and rules, not new kinds.

C3 A compositional semantics (Section 7). Well-formed subgraphs with boundaries are morphisms of a free hypergraph category [35, 136]. Behaviour is a hypergraph functor into spans of transition systems [193]. As a result a self-contained subgraph mounts at any compatible boundary, and replacing it is sound under trace refinement.

C4 One traversal primitive (Section 8). Typed message passing over incidences, with determinism, incremental maintenance [46] and coordination-freedom [16] results. Dependency resolution, navigation, rendering, lifecycle planning, capability resolution, effect and flow analysis and code-push closure are instances, and the same algebra compiles into data stores.

C5 A change model (Section 9). Double-pushout patches in an adhesive category [102, 210] as transactions, commuting independent patches, refs and proposals, graph diffs as the unit of review, and code push as a region release with a closure computation and epoch-fenced activation.

C6 A language, a kernel, a protocol and a reference realization (Sections 10 to 13). RGL's four sublanguages and phases; an integration with the Kotlin compiler in which the graph supplies the facts that thirteen optimisations need, with the conditions under which each is sound; a data-oriented kernel with stated complexity; a wire protocol in which reads are traversals, watches are streams of Z-set deltas with cursors, changes are sessions with effect identity, and authorities federate through signed views filtered by labels; the capability profile a host must meet; a realization in Kotlin, with compiler plugins and coroutines, on Cloudflare, Google Cloud and Postgres, worked through end to end, on paper, for one feature; and four further stacks: the reference stack at large scale, a Chinese cloud stack, an alternate public cloud stack and a corporate stack hosted internally.

C7 Precedents, problems, scope and programming model (Sections 4 and 16 to 19). Lessons from game engines, with Unreal Blueprints as the gold standard, and from Kubernetes, actors, TLA$^+$, statecharts and React; twenty recurring problems of multi-runtime systems, each mapped to the mechanisms that address it; eleven classes of systems mapped onto the model, from chat to scientific simulation and neurosymbolic AI; and a division of labour between people and agents, with compilers, test and analysis tools and agents working on one graph, and the scale hypothesis stated so that it can be rejected.

C8 Evidence and a plan (Sections 14, 20 and 22). A mapping of Reaktor's forty modules and tools onto RG, and of what they gain when connected; a mechanical study of the 140 published design graphs; a survey of 283 features of Reaktor and five systems built on it; two bounded protocol models and five TLA$^+$ templates checked with TLC; fifteen hypotheses with rejection criteria; and a twelve-month plan built on them.

Organisation. Section 2 describes the current runtime, Section 3 the background the paper uses, and Section 4 the precedents, game engines first. Sections 5 to 9 define the model, its types, its semantics, its traversals and its change model. Sections 10 to 13 present the language and its compiler, the kernel, the graph protocol and the reference realization, and Section 14 maps Reaktor's modules onto the graph. Section 15 covers assurance, Section 16 the recurring problems of multi-runtime systems, and Section 17 the classes of systems in scope. Sections 18 and 19 turn to programming at scale and to tools, agents and teams. Sections 20 to 22 give the evaluation, related work and the plan.

Figure 1 shows one chat feature across the six layers; the rest of the paper defines each element of that figure. Figure 2, on two facing pages, draws a whole platform the same way: two phones and a browser, the edge network, a cloud region, a managed database and external services, with the region that can be pushed as code highlighted.

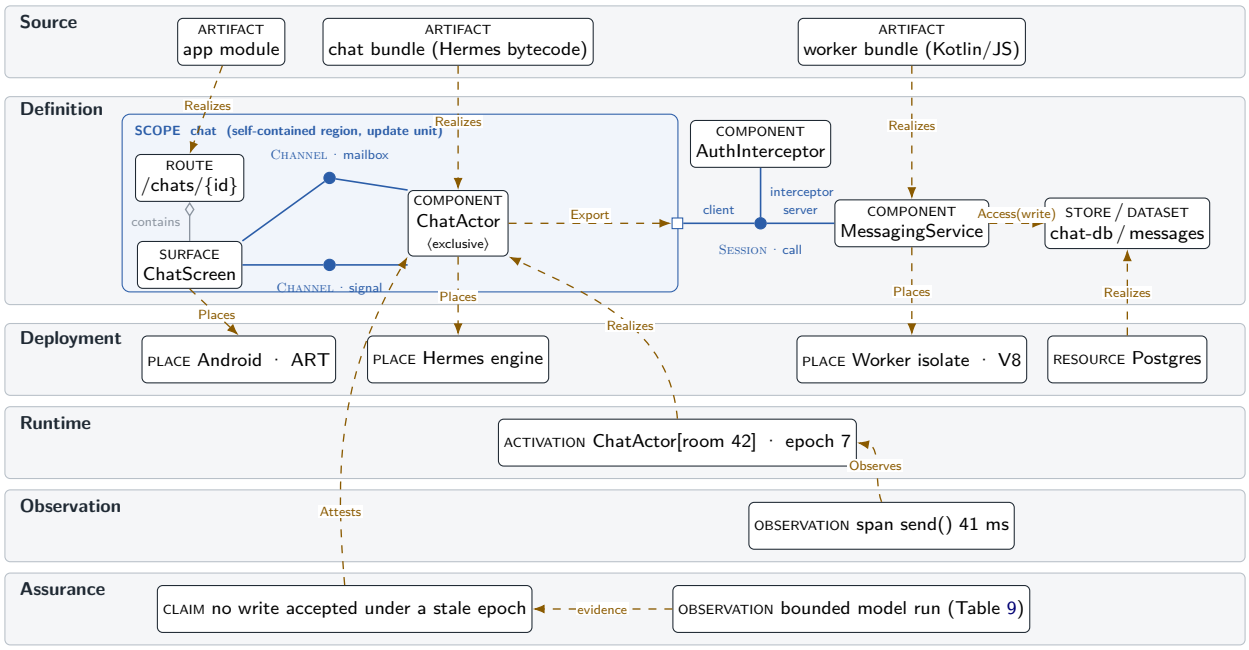


Figure 1: One chat feature in the six layers of RG. Blue lines are *connections*, the incidences of an executable hyperedge (filled dot). Brown dashed arrows are *assertions*, which relate elements and often cross layers. Grey diamonds mark containment. The Scope `chat` is self-contained: its only outward incidence passes through a boundary port. The whole region can therefore be mounted elsewhere or shipped to a Hermes engine as one bundle. The layers are fibres of one typed graph, not separate stores.

2 The current runtime and what it cannot express

2.1 The current runtime in brief

Reaktor is a Kotlin Multiplatform framework that also runs C++ and interoperates with TypeScript. Its graph has three modules. `reaktor-graph-port` is a domain-free kernel. It defines `PortGraph` and `PortNode`; `ProviderPort<F>`, which holds a live implementation object and serves many consumers; `ConsumerPort<F>`, which holds exactly one edge; and a binary `Edge<C>` from consumer to provider. A port is identified by its owner, a string key and a `Type`, which is a simple class name plus an optional `KClass`. Calls through a port pass through *interceptors*, and any element can carry typed *attachments*. `reaktor-graph-runtime` adds `Graph`, which bundles dependency injection over Koin, lifecycle, a coroutine scope and navigation. It also adds a sealed `Node` hierarchy (`BasicNode`, `ControllerNode`, `RouteNode`, `ContainerNode`, `ActorNode`). `autoWire` matches consumers to providers by type and then key, and falls back to the parent graph's injection scope. `RouteNode` binds a path pattern to a typed payload and owns `NavigationEdges`, the only subclass of `Edge`. Each graph keeps a back stack with push, replace, pop and return-with-result. `ServiceNode` registers one provider port per operation of a typed `Service` contract, typed by the string `op:<operation>:<request>-><response>`. `ActorNode` has a bounded mailbox, exclusive state and a supervision directive. `reaktor-graph` renders the graph with Compose: it takes the top back-stack entry and calls the attached node's `Content()`, recursing into containers.

A production social application on the current runtime composes 11 graphs and 3 containers over three levels of nesting, 23 routes and about 88 nodes and 300 ports. On its current branch, mobile clients mount a per-user session graph with 23 routes and 68 nodes. Each Cloudflare worker builds its own graph of its `ServiceNodes` and serves its shape at `/_reaktor/shape`. A separate structural IR, `GraphDefinition`, already exists. It has nested regions, nodes with an open kind code, ports with profile, contract and polarity (`Offer`, `Require`, `Neutral`), and n-ary relations whose incidences carry a role and an ordinal. The documentation publishes 140 design graphs in this format, and the test suite validates and round-trips 30 of them. Nothing connects the IR to the runtime or the editors [291, 292].

2.2 Requirements

Table 1 lists the seventeen requirements this paper answers and the status of each in the current runtime, with source evidence.

2.3 Five root causes

The gaps share five causes, and each shapes the design.

(1) **One relation shape.** The only runtime edge is a pair (consumer, provider). Navigation, service calls, authoriza-

	Requirement	Now	Evidence in the current runtime	RG mechanism
R1	Mount self-contained subgraphs anywhere	implemented	Nested graphs and containers; but edges cross scopes freely and DI resolves through parents, so boundaries are implicit	Scopes, boundary ports, Export; Theorem 3
R2	Navigation as traversal	partial	Routes are nodes; back stacks per graph; deep links reach only plain-payload routes; cycles need imperative back-patching	Transition with statechart semantics; Section 8.3
R3	Recursive rendering of containers and screens	implemented	Fixed interpreter; custom designs copy 530 lines of framework containers	Surface slots; containment folds
R4	Client/server contracts that cannot drift	partial	Shared Kotlin types and schema fingerprints; route strings duplicated and unchecked; servers dispatch outside the graph	Contract fingerprints; Session; realization by placement
R5	Bolt-on auth, telemetry and similar	partial	Two interceptor systems; secured ports carry requirements that `connect` never checks	Interceptor roles; aspects as rewrite rules (Section 9)
R6	Powerful, general traversals	partial	Three visitors in both repositories; they mutate, return nothing, and breadth-first drops the exit callback	One primitive with theorems (Section 8)
R7	First-class DI and framework interop	implemented	Type-then-key matching; ambiguity undetected (two providers of one port bind arbitrarily); missing providers fail at call time	Bind resolution as a lattice fixpoint with findings
R8	Data-oriented representation	absent	Pointer graph; 64-bucket maps scanned per access; whole graph re-snapshotted per navigation	Columnar C-set kernel (Section 11)
R9	FFI, publish/subscribe, queues	absent	`PUBSUB`, `QUEUE`, `PEER`, `WORKFLOW` transports declared, never dispatched; FFI is a Hermes "hello"	Channel, Session; FFI as a realization
R10	Structured concurrency as part of the graph	partial	`SupervisorJob` and dispatchers fixed in classes; `GlobalScope` escapes; lifecycle effects fire on rejected transitions	Runtime containment is the job tree; lifecycle plans
R11	Code push	absent	Kernel pulls in Koin, Ktor and the database; `Uuid.random` needs `crypto`; dispatcher needs `setTimeout`; `KClass` identity cannot match across bundles	Platform-free kernel; capability ports; region release
R12	Data layer in the graph	absent	Key–value object store; typed SQL DSL unused; tables found by regex heuristics	Store, Dataset, References, Access
R13	Capability layer	absent	"Capability" names lifecycle mixins; the availability runtime is design only	Place-provided capability ports; support lattices
R14	One graph for the whole platform	partial	Island shapes, a cloud snapshot and a parsed topology: three infrastructure graphs with no shared key	Six layers, one store; projections (Section 14)
R15	A language for arbitrary logic in a subgraph	absent	Kotlin only; no code entries, no distributed operations	RGL (Section 10)
R16	Formal guarantees inside the system	absent	No specifications; a single ordered truth enum; grep-based architecture tests	Claims and evidence as elements; generated specs
R17	Code and ecosystem as database and logic; transactional edits	partial	Definitions exist only as code; edits mutate immediately; no revisions	Graph store, patches, diffs; a realization recorded for every element

Table 1: Requirements and their status in the current runtime. Evidence is from the Reaktor repository at `7f318a55` and from the application's repository at `f6c19f13`.

tion and actor references are all squeezed into it, and semantic kinds are recognised by port key names such as `routeBinding`. A topic with many subscribers, a session, a join or a grant to a principal over a resource cannot be stated as one relation.

(2) **Objects instead of data.** The graph is a web of mutable objects. Node identifiers are `Uuid.random()` per process. Wiring order follows them, so dependency resolution can differ between runs. There are no revisions, digests or patches.

(3) **Behaviour fixed in classes.** Lifecycle, coroutine scopes, the navigation interpreter and the rendering interpreter are methods of `Graph` and `Node`. New semantics therefore require subclasses rather than traversals.

(4) **Platform coupling.** Every `Graph` needs a dependency-injection adapter, and the runtime depends on Ktor and the database module. Identity and scheduling assume platform APIs. This blocks running the graph inside an embedded JavaScript engine, and hence code push.

(5) **No model of change or evidence.** Edits mutate immediately and emit per-entity events. Definitions, runtime shapes, cloud inventories and evidence are separate artefacts with no shared identity.

RG addresses (1) with typed hyperedges, (2) with a columnar store of immutable revisions, (3) with traversals, (4) with a platform-free kernel whose platform access goes through capability ports, and (5) with patches and assurance elements.

The same causes at platform scale. A survey of 283 features of Reaktor and five production systems built on its current runtime (Section 20.2) found these causes repeated across the platform as five patterns:

(i) *One system, many models.* A single developer tool held up to seven models of the same system, joined by names; one

product declared its deployment topology in seven hand-kept places.

(ii) *Identity that does not survive a run.* Spans, crashes and tool views were joined by label.

(iii) *Effects lost or misreported.* Post-commit effects were fired without an outbox, and expired deadlines were reported as failures.

(iv) *Actions without principals.* Approvers were free strings, and self-approval was possible on every path that changes tooling.

(v) *Authority written in code.* Checks were opt-in per handler, and the requirements of secured ports were never enforced.

Each pattern is one of the five causes seen from outside the graph.

3 Preliminaries

The design draws on four bodies of theory and two bodies of engineering practice. This section fixes the notions the paper uses, in the form it uses them, and says where each is used. Appendix E gives a longer primer with worked examples, and Appendix D a reading plan.

3.1 Graphs, hypergraphs and networks

A *directed graph* has vertices, edges, and source and target maps from edges to vertices. A *hypergraph* lets one edge join any number of vertices [30]. Any hypergraph can be drawn as a bipartite *incidence graph*, with vertices and hyperedges as two sorts of node and one edge per incidence. Kernels store a hypergraph this way, and graph algorithms run on it.

- In a *port graph*, vertices own named ports and edges attach to ports, so a connection says not only which components meet but at which interface.
- Network science adds measures over large graphs [26, 265]: degree, centrality, communities and percolation. Multilayer networks keep several edge types over one vertex set [197].
- A RG is a port hypergraph whose incidences carry a role and a position, plus a forest of containment (Section 5). Its incidence graph is where network measures apply: centrality estimates blast radius, and communities propose regions.

Figure 3 draws one session in the three forms.

3.2 Categories, presheaves and rewriting

A *category* has objects and composable arrows; a *functor* maps one category to another, preserving composition [137, 221].

- *C-sets.* For a small category C, a functor $X : C \to \mathsf{Set}$ is a *C-set*, also called a copresheaf. Each object of C becomes a set (a table) and each arrow a function (a column of foreign keys). Graphs are the C-sets over the category with two objects E and V and two arrows $s, t : E \to V$. Morphisms of C-sets are natural transformations, and Set^C computes limits and colimits table by table [280].
- *Typing.* A graph typed over a type graph T is a morphism into T. The category of such graphs is again one of presheaves, $\mathsf{Set}^{\int T}$.
- *Gluing.* A *pushout* glues two objects along a common part.
- *Rewriting.* A *double-pushout* (DPO) rule $L \leftarrow K \rightarrow R$ deletes the image of $L \setminus K$ and glues in R along K [102]. With an injective match, it applies exactly when nothing that remains would be left dangling. In an *adhesive* category, pushouts along monomorphisms behave like unions of sets [210]. There, rewriting has the Church–Rosser, parallelism and concurrency theorems that Section 9 uses.
- *Open systems.* A cospan $J \rightarrow G \leftarrow K$ is a system G with input boundary J and output boundary K, and cospans compose by pushout. *Hypergraph categories* are the algebra of such wiring [136]: free ones correspond exactly to cospans of hypergraphs with discrete interfaces [35]. A functor into spans of transition systems then assigns behaviour compositionally [193] (Section 7).

3.3 Order, fixpoints and incremental computation

- *Lattices.* A *lattice* is a partial order with binary joins and meets [82]. On a complete lattice, every monotone map has a least fixpoint (Knaster–Tarski). On a lattice of finite height it is reached by iterating from the bottom element. *Chaotic iteration* reaches the same fixpoint in any fair order [75]: this is why a traversal's result does not depend on schedule or partition (Theorem 6).
- *Semirings.* A semiring $(S, \oplus, \otimes)$ generalizes path problems [250]: Boolean for reachability, min-plus for the shortest route, max-plus for the worst-case latency of an acyclic call chain, max-min for capacity.
- *Datalog* evaluates recursive rules to their least fixpoint, with negation allowed only between strata [4]. Lattice-valued Datalog generalizes it [232]. Program analyses are the classic use [266].
- *Incremental computation.* A *Z-set* is a multiset with integer multiplicities, so an insertion is $+1$ and a deletion -1. DBSP maintains a query over Z-sets under deltas; for linear operators the cost is proportional to the change [46].
- *Coordination.* The CALM theorem says a problem has a consistent, coordination-free distributed implementation exactly when it is monotone [16, 167].

- *Galois connections.* A Galois connection between two orders relates a requirement to the offers that meet it; Section 6.6 uses one.

3.4 Specification, refinement and model checking

A system is a state machine: an initial condition and a next-state relation. A behaviour is an infinite sequence of states. TLA$^+$ writes a specification as $Init \wedge \Box[Next]_v \wedge Fairness$ [214].

- *Safety and liveness.* A *safety* property says that nothing bad ever happens; a *liveness* property, that something good eventually does [12].
- *Refinement.* One specification *refines* another if each of its behaviours, mapped through a refinement mapping, is a behaviour of the other [1]. Composition is conjunction [2].
- *Model checking* explores every reachable state of a bounded instance [24]. Breadth-first exploration returns shortest counterexamples, and the bounded models of Table 9 work this way.
- *Other methods.* Alloy checks first-order structural constraints within bounds [178]. Proof assistants such as Lean prove properties for all sizes [85].

Appendix C.1 gives a complete TLA$^+$ module for the Session call family.

3.5 Machines: memory, layout and cost

Each level of the memory hierarchy is larger and at least an order of magnitude slower than the one above it: registers, caches, main memory, solid-state storage. The network lies beyond them [100, 168]. Data moves between caches and memory in lines of typically 64 bytes, so cost is dominated by how many lines a computation touches.

- *What hardware rewards.* Sequential access lets hardware prefetch. Following pointers defeats it.
- *Layout.* Storing each field in its own array (*structure of arrays*) rather than objects with all fields (*array of structures*) puts the data a loop needs in consecutive lines.
- *Sparse structure.* Compressed sparse row (CSR) stores sparse incidence the same way [194].
- *Handles.* Generational handles replace pointers that may dangle.

This is *data-oriented design* [5, 130], and it is how the kernel stores the graph (Section 11). On the JVM and in JavaScript engines the same layout also avoids allocation and garbage-collection pressure, through primitive and typed arrays.

Across a multi-runtime system the same arithmetic holds at a larger scale [86]. Each step below typically costs one or more orders of magnitude more than the one before it:

- an in-process call (nanoseconds);
- a call across a native boundary;
- a call between processes;
- a remote call within a region (around a millisecond);
- a call across continents (tens to hundreds of milliseconds).

Placement and batching therefore dominate end-to-end cost. This is why RG lets location be abstracted but never latency (Section 6).

3.6 Distributed-systems reality

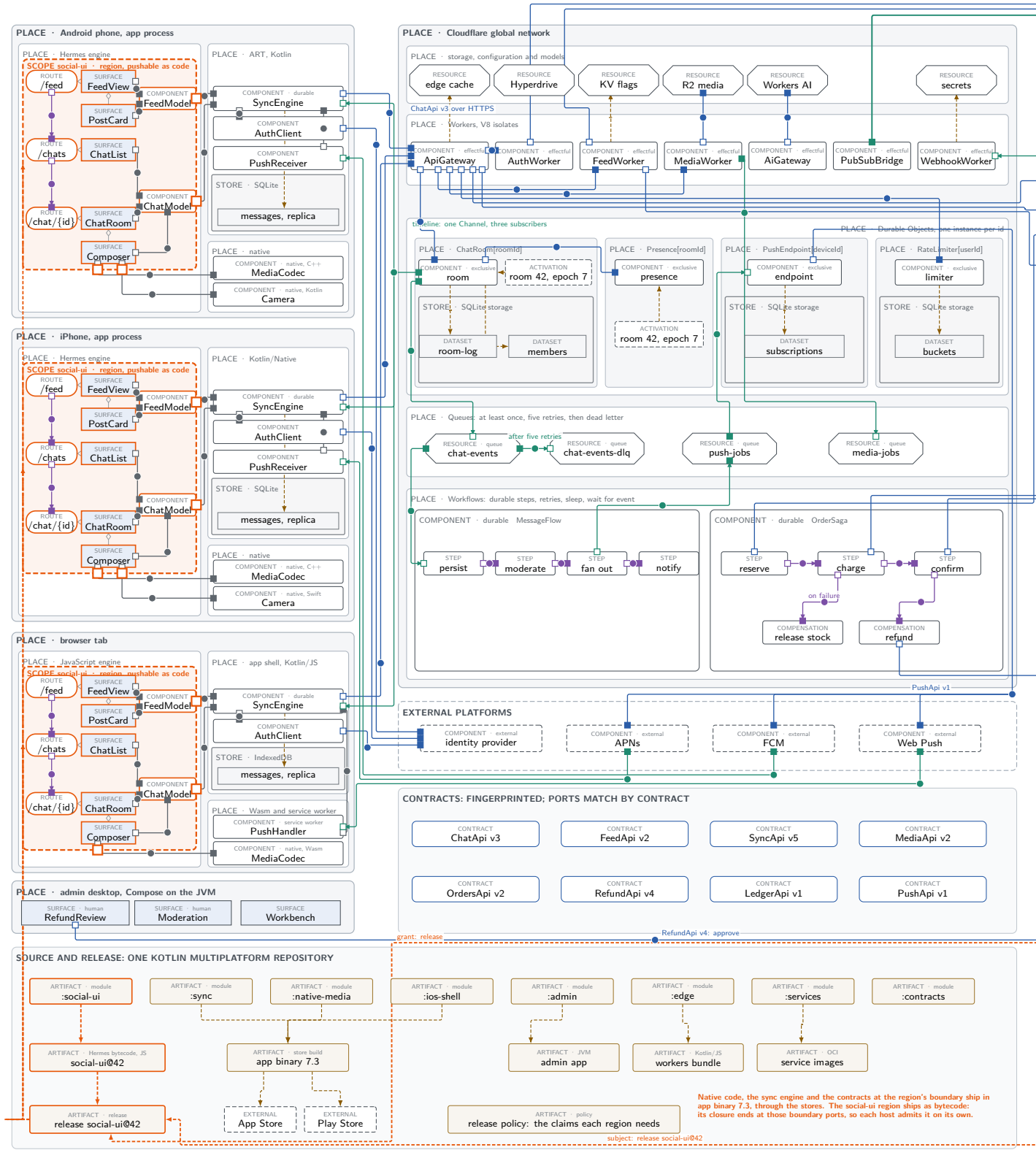
Components fail independently, and messages may be delayed, lost, duplicated or reordered [48, 230].

- *Impossibility.* In an asynchronous system, no deterministic protocol reaches consensus if even one process may crash [134]. Under a network partition, a service must choose between consistency and availability [144].
- *Timeouts.* A timeout cannot tell a slow peer from a dead one, so a call that timed out has an *unknown* effect.
- *Exactly once.* Delivery can be at most once or at least once. "Exactly once" is at-least-once delivery plus deduplication by a stable effect identity, an idempotency key, at the place where the effect commits [166].
- *Leases and fencing.* A lease holder may pause past its expiry. Only the authority that stores the data can reject a stale holder, by comparing a monotonically increasing epoch, or *fencing token* [155, 199].
- *Consensus.* Paxos and Raft build replicated authorities that issue such epochs [213, 272].
- *Cross-service changes.* Independent services rarely share ACID transactions [166]. A change that spans them commits an outbox record with the local effect, or runs as a saga with compensations [140].
- *Coordination-free convergence.* State that only grows by joins converges without coordination: CRDTs [310].
- *Time.* Logical clocks order events consistently with causality, and consistent snapshots capture a global state [56, 211].

Figure 4 draws the case that leases alone get wrong. Sections 7, 9 and 13 build each of these into families, outcomes and activation.

4 Precedents: what game engines already know

Few kinds of software are as complex, or as fast, as a modern game. Within one frame, 16.7 ms at sixty frames per second, an engine simulates physics, animates characters, runs their AI, mixes audio, replicates state to remote players and renders the scene. A large game is built by hundreds of people, and most of them do not write C++: designers,


PLACE · Android phone, app process
PLACE · Hermes engine
SCOPE social-ui · region, pushable as code
ROUTE /feed
SURFACE FeedView
COMPONENT FeedModel
SURFACE PostCard
ROUTE /chats
SURFACE ChatList
COMPONENT ChatModel
ROUTE /chat/{id}
SURFACE ChatRoom
SURFACE Composer
PLACE · ART, Kotlin
COMPONENT · durable SyncEngine
COMPONENT AuthClient
COMPONENT PushReceiver
STORE · SQLite
messages, replica
PLACE · native
COMPONENT · native, C++ MediaCodec
COMPONENT · native, Kotlin Camera
PLACE · iPhone, app process
PLACE · Hermes engine
SCOPE social-ui · region, pushable as code
PLACE · Kotlin/Native
COMPONENT · durable SyncEngine
COMPONENT AuthClient
COMPONENT PushReceiver
STORE · SQLite
messages, replica
PLACE · native
COMPONENT · native, C++ MediaCodec
COMPONENT · native, Swift Camera
PLACE · browser tab
PLACE · JavaScript engine
SCOPE social-ui · region, pushable as code
PLACE · app shell, Kotlin/JS
COMPONENT · durable SyncEngine
COMPONENT AuthClient
STORE · IndexedDB
messages, replica
PLACE · Wasm and service worker
COMPONENT · service worker PushHandler
COMPONENT · native, Wasm MediaCodec
PLACE · admin desktop, Compose on the JVM
SURFACE · human RefundReview
SURFACE · human Moderation
SURFACE Workbench
PLACE · Cloudflare global network
PLACE · storage, configuration and models
RESOURCE edge cache
RESOURCE Hyperdrive
RESOURCE KV flags
RESOURCE R2 media
RESOURCE Workers AI
RESOURCE secrets
ChatApi v3 over HTTPS
PLACE · Workers, V8 isolates
COMPONENT · effectful ApiGateway
COMPONENT · effectful AuthWorker
COMPONENT · effectful FeedWorker
COMPONENT · effectful MediaWorker
COMPONENT · effectful AiGateway
COMPONENT · effectful PubSubBridge
COMPONENT · effectful WebhookWorker
timeline: one Channel, three subscribers
PLACE · Durable Objects, one instance per id
PLACE · ChatRoom[roomId]
COMPONENT · exclusive room
ACTIVATION room 42, epoch 7
STORE · SQLite storage
DATASET room-log
DATASET members
PLACE · Presence[roomId]
COMPONENT · exclusive presence
ACTIVATION room 42, epoch 7
PLACE · PushEndpoint[deviceId]
COMPONENT · exclusive endpoint
STORE · SQLite storage
DATASET subscriptions
PLACE · RateLimiter[userId]
COMPONENT · exclusive limiter
STORE · SQLite storage
DATASET buckets
PLACE · Queues: at least once, five retries, then dead letter
RESOURCE · queue chat-events
after five retries
RESOURCE · queue chat-events-dlq
RESOURCE · queue push-jobs
RESOURCE · queue media-jobs
PLACE · Workflows: durable steps, retries, sleep, wait for event
COMPONENT · durable MessageFlow
STEP persist
STEP moderate
STEP fan out
STEP notify
COMPONENT · durable OrderSaga
STEP reserve
STEP charge
STEP confirm
on failure
COMPENSATION release stock
COMPENSATION refund
PushApi v1
EXTERNAL PLATFORMS
COMPONENT · external identity provider
COMPONENT · external APNs
COMPONENT · external FCM
COMPONENT · external Web Push
CONTRACTS: FINGERPRINTED; PORTS MATCH BY CONTRACT
CONTRACT ChatApi v3
CONTRACT FeedApi v2
CONTRACT SyncApi v5
CONTRACT MediaApi v2
CONTRACT OrdersApi v2
CONTRACT RefundApi v4
CONTRACT LedgerApi v1
CONTRACT PushApi v1
grant: release
RefundApi v4: approve
SOURCE AND RELEASE: ONE KOTLIN MULTIPLATFORM REPOSITORY
ARTIFACT · module :social-ui
ARTIFACT · module :sync
ARTIFACT · module :native-media
ARTIFACT · module :ios-shell
ARTIFACT · module :admin
ARTIFACT · module :edge
ARTIFACT · module :services
ARTIFACT · module :contracts
ARTIFACT · Hermes bytecode, JS social-ui@42
ARTIFACT · store build app binary 7.3
ARTIFACT · JVM admin app
ARTIFACT · Kotlin/JS workers bundle
ARTIFACT · OCI service images
ARTIFACT · release release social-ui@42
EXTERNAL App Store
EXTERNAL Play Store
ARTIFACT · policy release policy: the claims each region needs
Native code, the sync engine and the contracts at the region's boundary ship in app binary 7.3, through the stores. The social-ui region ships as bytecode: its closure ends at those boundary ports, so each host admits it on its own.
subject: release social-ui@42
Kinds
COMPONENT component
SURFACE surface
ROUTE /route
DATASET dataset
RESOURCE resource
PRINCIPAL principal
ARTIFACT artifact
ACTIVATION activation
OBSERVATION observation
CLAIM claim
CONTRACT contract
STORE
PLACE
Connections
Bind: in-process call
Session: call across places
Channel: messages, at least once
Transition: step or navigation
Ports, hubs
provided
required
hyperedge, one per connection; each line to a port is an incidence
assertion: access, realizes, derives, references, grant, attests, observes, places
Change
region that can be pushed as code: one scope, instantiated on three hosts
boundary port: an FFI adapter whose contract the installed binary fixes
release path, orange
containment: route to surface, surface to part. A box drawn inside a place is placed there.

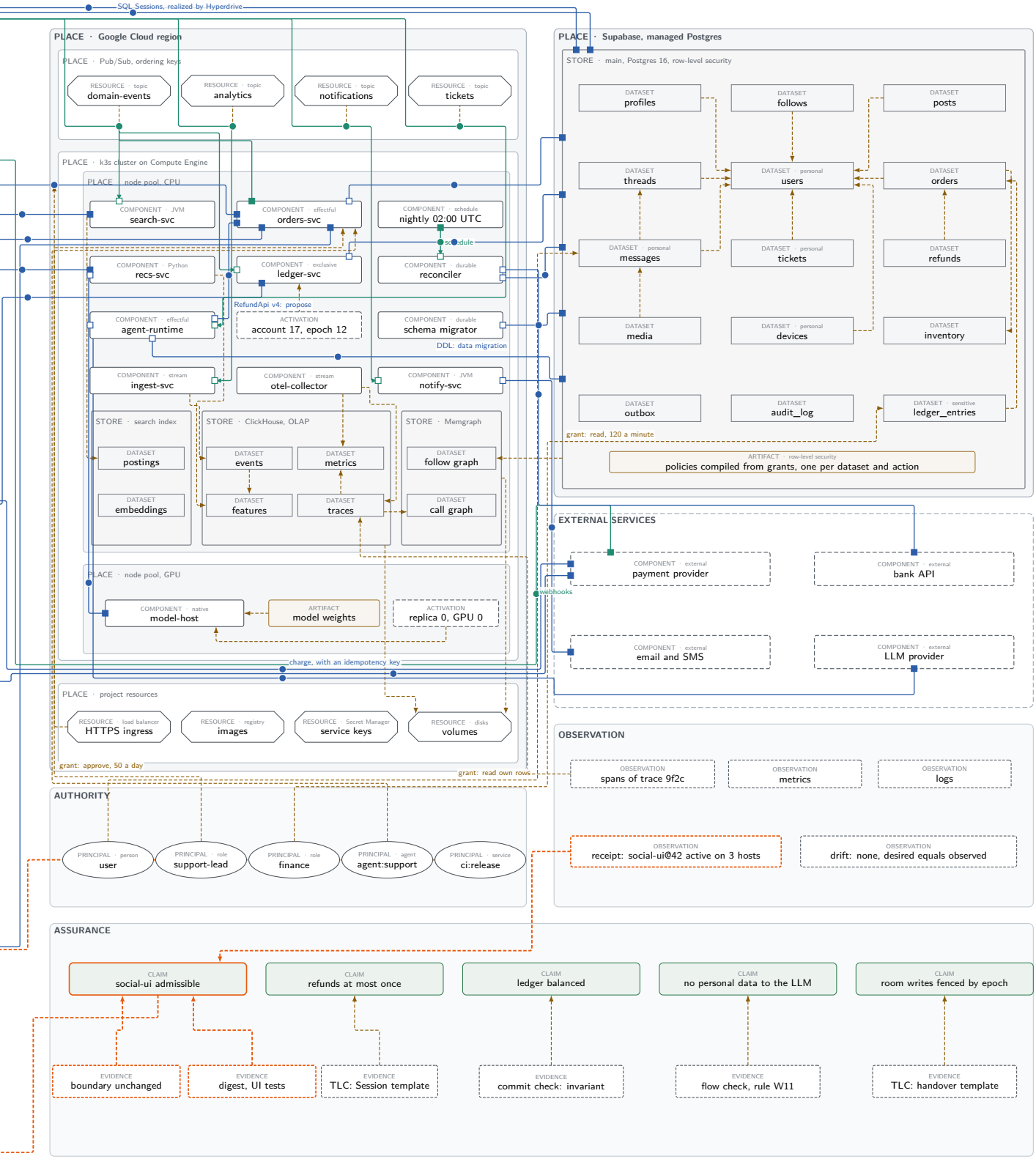
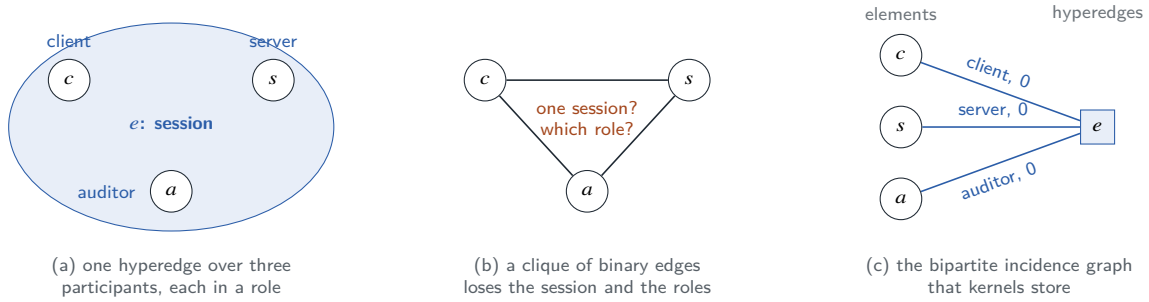


Figure 2: A whole platform as one Reaktor graph, on two facing pages from page 8: two phones and a browser, the edge network, a cloud region, a managed database and external services, from source modules to claims. The orange subgraph can be pushed as code: the scope `social-ui`, instantiated in the Hermes engines of both phones and in the browser. Its binds leave it only through boundary ports, FFI adapters whose contracts the installed app binary provides, so its closure (Equation (3)) ends there and `release social-ui@42` replaces it under next-entry activation without a store release. The claim *social-ui admissible* gates the release, and a receipt records it. Changing `SyncApi` would pull `SyncEngine`, and so the app binary, into the closure. The graph has 229 elements, 198 ports, 100 connections with 205 incidences and 71 assertions. It illustrates the design; it is not a deployed system.



(a) one hyperedge over three participants, each in a role

(b) a clique of binary edges loses the session and the roles

(c) the bipartite incidence graph that kernels store

Figure 3: One session among a client, a server and an auditor, drawn three ways. Each incidence of (c) carries a role and a position, so nothing that (a) states is lost (Appendix E.1).

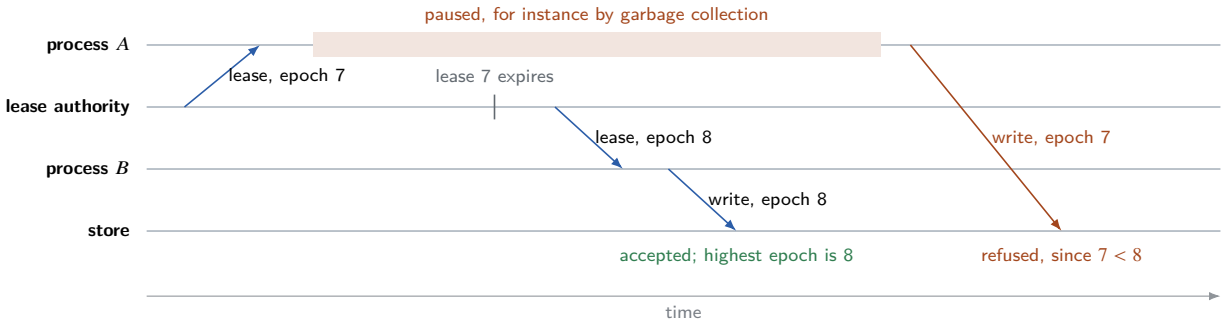


Figure 4: Fencing. A lease holder pauses past its expiry, wakes and writes with a stale epoch. Only the store that holds the data can refuse that write, by comparing epochs. Without the check both writes are accepted, which is the ownership-handover counterexample of Table 9.

animators, technical artists and sound designers change the game every day. This section asks how such software is organised. Its organisation is the closest working precedent for what RG proposes, and the engines arrived at it before language models could write code. Unreal Engine is our main example. Section 4.5 returns to the other systems named in Section 1.

4.1 Unreal Engine: a typed core, reflection and graphs

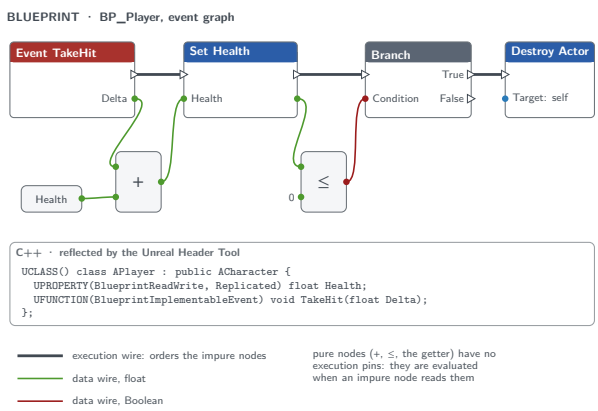
Unreal Engine 4 was released in March 2014 [325], with Blueprints as its visual scripting system [110]. Blueprints succeeded Kismet, in which designers of Unreal Engine 3 wired sequences of actions to game events [128], and the Blueprint compiler still carries Kismet's name [113]. Five layers make up the result.

A native core. The engine, and the parts of a game that must be fast, are written in C++. Rendering, physics, animation evaluation, networking and the inner loops of gameplay run there.

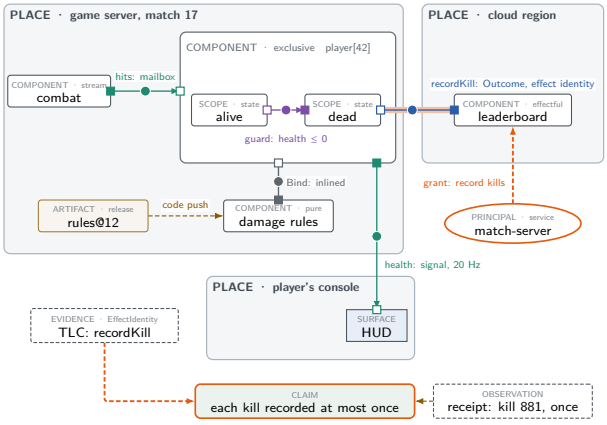
Reflection. Programmers annotate C++ declarations with macros: `UCLASS` for a class, `UPROPERTY` for a property and `UFUNCTION` for a function. The Unreal Header Tool parses the annotations and generates reflection code [124]. One declaration then serves many consumers: garbage collection, serialization, the editor's property panels, run-time type information, network replication and Blueprints. Specifiers on each declaration decide what Blueprints may do with it: `BlueprintCallable` exposes a function to graphs, `BlueprintImplementableEvent` lets a Blueprint supply a function's body, and `BlueprintReadWrite` exposes a property for reading and writing.

Blueprints. A Blueprint is a graph of nodes with typed pins [114]. *Execution* pins order the nodes that have effects. *Data* pins carry values, and their wires are typed: the editor refuses a wire between incompatible pins. *Pure* nodes have no execution pins and are evaluated when a node that reads them runs. A Blueprint class can derive from a C++ class, so designers extend what programmers expose; C++ can also declare events whose bodies a Blueprint supplies. The Blueprint compiler turns a Blueprint into a generated class, merges its event graphs into one, and emits bytecode for the engine's script virtual machine [113]. Unreal Engine 4 could also generate C++ from Blueprints when cooking, to remove the virtual machine's overhead; Unreal Engine 5 removed that option [109, 121]. Figure 5a shows a small event graph.

Many graph languages, one object model. Materials are expression graphs translated to HLSL and compiled into shaders [120]. Animation has blend graphs and state machines. AI has behaviour trees and StateTree, a hierarchical state machine that combines the selectors of behaviour trees with states and transitions [126]. Visual effects, audio and procedural content have graph editors of their own. Each language has its own semantics (dataflow, state machine, tree),



(a) A Blueprint event graph, redrawn. Thick wires carry execution and order the impure nodes; thin wires carry data and are typed by colour; the pure nodes have no execution pins. The variable and the event come from a C++ class through reflection.



(b) The same rule in RG. Execution becomes a TRANSITION inside an exclusive component, data becomes BIND and CHANNEL, and a replicated property becomes a signal. Orange marks what a graph of one game process cannot state: the SESSION to a service in another place, its grant, and a claim checked by TLC.

Figure 5: Blueprints and RG side by side. The precedent supplies typed pins, the split between execution and data, and a native core behind reflection; RG keeps them and extends the graph past the boundary of one process.

and all of them share one reflected object model, one asset system and one editor.

A runtime built for scale. Several subsystems carry the load when a world grows:

- *MassEntity*, a framework for data-oriented computation, stores data-only fragments grouped by archetype in chunks and runs stateless processors over them [119].
- *Replication* starts from a server that holds the authoritative state. Properties marked for replication are pushed to clients, remote procedure calls are marked reliable or unreliable, and relevancy, decided per connection, determines what each client receives [125].
- *The Gameplay Ability System* gives an actor a component that holds its granted abilities, its gameplay tags and its attributes, and applies effects that change attributes and tags in a way compatible with network prediction [116].
- *World Partition* keeps one persistent level, divided into grid cells that stream in and out by distance [127]. With One File Per Actor, each actor is saved in its own file, which reduces contention for source-control locks [122].

Tools over the same model. The Asset Registry indexes assets, including those not loaded, together with their dependencies, and the Reference Viewer draws the resulting graph [111]. Unreal Insights records trace events, selected by channel, for live or later analysis [117]. Live Coding rebuilds C++ and patches it into the running editor or game on desktop platforms [118].

4.2 Why it works

Five decisions explain why this organisation produces software that is both complex and fast.

1. *Composition is separate from computation.* Graphs compose and the native core computes. Epic advises keeping heavy and frequent work in C++, and its programmers turn complex Blueprint logic into new C++ nodes [115]. The graph's overhead is then paid per event rather than per entity.
2. *Types sit at the pins.* The editor rejects a wire between incompatible pins. People who do not program can therefore compose with less risk: a class of errors moves from run time to edit time, although a graph can still fail at run time in other ways.
3. *One reflected model serves every tool.* The editor, the serializer, the garbage collector, the network layer, the profiler and the cooker read the same metadata, so they agree with one another.
4. *Effects are ordered explicitly; data flows implicitly.* Execution wires fix the order of effects. Pure data is computed on demand.
5. *Iteration is fast.* An edit compiles in seconds, the game runs inside the editor, and Live Coding patches native code without a restart.

Blueprints are the gold standard for what RG attempts. Epic shipped them in 2014, years before language models could write code. Their users were mixed teams of designers, artists and programmers. Their lesson is that a typed graph over a native core lets people with different skills change one complex, fast system while a class of errors is caught before the game runs. Agents built on language models are one more kind of collaborator with the same need: they also work best

In Unreal Engine	In RG	What RG adds
C++ core behind reflection; the Unreal Header Tool	Kotlin and C++ bodies behind ports; symbol processing and the K2 plugin extract contracts (Section 13.4)	contract fingerprints that hold across runtimes
Typed pins; wires checked in the editor	typed ports; W3–W5 in the IDE and workbench	hyperedges with several parties and roles
Execution pins versus data pins; pure nodes	Transition versus Bind and Channel; the pure profile	durable transitions; sessions with outcomes
A Blueprint class extends a C++ class	a graph-owned scope over source-owned components	DPO patches; mounting at boundaries (Corollary 4)
Graph languages for materials, animation, AI, audio and procedural content	profiles, derived kinds and domain modules over Σ_{std}	one signature; traversals shared by all domains
Blueprint compiler and virtual machine; materials compiled to shaders	graph-owned regions compiled to Kotlin; traversals compiled into stores	optimisation with the whole graph (Section 10.7)
Replication: authority, replicated properties, reliable and unreliable calls, relevancy	exclusive components with epochs; signal channels with QoS; sessions; interest as a traversal	fencing at stores that check epochs; effect identity elsewhere
Gameplay Ability System: granted abilities, gameplay tags	Grants, labels and facets	no self-approval (A5); label flow (W11)
Mass Entity: fragments, archetypes, processors	the columnar kernel (Section 11)	the system graph stored the same way
World Partition: streamed cells; a file per actor	regions and keyed places; one row per element	patches that commute and merge
Asset Registry: dependencies for cooking and chunks	impact and closure traversals	closures across runtimes; code push
Unreal Insights: trace channels	Observations joined by identity	coverage claims
Live Coding	region release (Section 9.5)	admission, fencing and rollback

Table 2: Unreal Engine's ideas and their counterparts in RG.

on typed, checkable units with immediate feedback (Section 19).

4.3 Where the analogy stops

Blueprints describe one game. A multi-runtime system needs more.

- *One process, one world.* A Blueprint describes behaviour inside one game process, and replication extends its state to the clients of that game. Matchmaking, accounts, stores, telemetry pipelines, analytics and live operations run outside the engine, in other languages and tools, with no shared model. That outside is the subject of this paper.
- *Changes are files.* Blueprints and levels are binary assets. They cannot be merged as text, so source control locks them on checkout [123], and Epic's own guidance notes that C++ is easier to diff and merge because it is text [112]. One File Per Actor exists to reduce the resulting contention [122]. In RG the unit of change is the element, and patches that touch different elements commute and merge (Theorem 10).
- *The implementation is the semantics.* A Blueprint means what its virtual machine does. There is no specification to check, no claim that goes stale when code changes, and no notion of an effect whose outcome is unknown.
- *Authority is coarse.* Who may change what is a matter of file permissions and review practice. There are no grants on elements and no rule that separates proposing a change from approving it.
- *Graphs do not scale without limit.* Large event graphs are hard to read, and they run more slowly than native code [115]. RG keeps graphs at the level of composition (components, connections and places) and leaves algorithms in code.
- *Two execution paths are expensive.* Nativization gave Blueprints a second path, through generated C++, and Unreal Engine 5 removed it [109]. RGL also has two paths, the generic runtime and specialised regions, and tests their equivalence by running one conformance suite on both (Section 10.7).

Other engines show the same pattern. Unity added visual scripting and a data-oriented entity system [341, 342]. Godot removed its visual scripting language from Godot 4 [225]. Our reading is that visual graphs succeed where they compose a strong, typed core, and struggle where they duplicate a general-purpose language without one. RG is designed to be the first kind.

4.4 From Blueprints to RG

Table 2 maps each idea above to its counterpart in RG and states what RG adds. Figure 5b draws the event graph of Figure 5a in RG notation. Execution becomes a Transition, data becomes Bind and Channel, and the replicated health value becomes a signal. The figure then continues past the game process, where no engine graph reaches: a Session to a leaderboard in another place, the grant that permits it, and a claim checked by TLC.

4.5 Other precedents

Kubernetes. The API server stores typed objects with a desired specification and an observed status. Controllers are control loops that move the current state towards the desired state. Custom resource definitions add new kinds with schemas; operators package the operational knowledge of one application as controllers; admission controllers and webhooks validate or modify an object before it is stored [335–338]. RG takes the open schema, reconciliation and admission. It adds typed relations where Kubernetes has labels and selectors, transactions over many objects, and semantics past the boundary of the container.

Actors. Hewitt's actors, Erlang/OTP and Orleans [20, 34, 169] give isolation, mailboxes, supervision trees and, in Orleans, activation on demand by identity. RG keeps each of these as a profile, a family or a facet. It rejects one actor idea: location transparency. A Bind never crosses a place, because a remote call fails and waits in ways that a local one does not (Section 6). The handover model also shows that actors need fencing at the stores they write (Table 34).

TLA$^+$. TLA$^+$ describes a system as a state machine, composition as conjunction and implementation as refinement [214], and it has been used on production designs at AWS [264]. RG generates specifications per family and region from the graph that the runtime executes, so the model and the system share one source. What can still diverge is a body that does not keep its contract, which tests and observations check.

Statecharts and Stately. Harel's statecharts add hierarchy, orthogonal regions and history to state machines [163], and SCXML standardises their execution [28]. XState implements statecharts and actors in TypeScript, and Stately provides a visual editor for such machines [320]. RG's routes and durable transitions are statecharts (Figure 13), and a graph-owned scope can likewise be edited visually or as text. What RG adds is that the machine is connected: its transitions call sessions, emit on channels and act under grants, all in one graph.

React and Compose. React and Compose made an interface a function of state, reconciled by identity: React by key, Compose by position in the composition, with optional keys [149, 242]. Compose also showed that a compiler plugin can deliver a programming model while the source stays ordinary Kotlin, and RGL's plugin follows it (Section 10.7). RG makes keyed reconciliation the meaning of its whole runtime layer, not only of the interface (Section 8.4).

Agent graphs. LangGraph runs long-running, stateful agents as graphs that mix deterministic steps with steps taken by a model, and offers durable execution and human-in-the-loop control [217]. It is a recent precedent for treating agent behaviour as a durable, inspectable graph. RG places such graphs inside the rest of the system, under grants and labels (Section 17.1).

4.6 What RG takes, and what it adds

From Unreal Engine, RG takes a typed graph over a native core, the split between execution and data, one reflected model for every tool, and data-oriented runtimes. From Kubernetes it takes desired and observed state, reconciliation, an open schema and admission. From actors it takes isolation, mailboxes, supervision and activation by identity; from TLA$^+$, specification, refinement and model checking; from statecharts, hierarchy and history, and one model in visual and textual form; from React and Compose, keyed reconciliation and a compiler plugin as the carrier of a programming model. Each precedent is deep in one domain and stops at its edge: the engine at the game process, Kubernetes at the container, actors at the mailbox, TLA$^+$ at the model, statecharts at the machine, React at the screen. RG joins them in one typed graph, so that the guarantees each one provides can be stated, and checked, across the boundaries between them as well.

5 The Reaktor Graph

5.1 Elements

Definition 1 (Reaktor graph)**.** A *Reaktor graph* is a tuple $G = (N, P, E, I, own, par, sc, edge, end, role, ord, kind, attr)$ where

- N, P, E and I are finite, pairwise disjoint sets of *nodes*, *ports*, *hyperedges* and *incidences*, and $X = N \uplus P \uplus E$ is the set of *elements*;
- $own : P \to N$ is total: every port has exactly one owner;
- $par : N \rightharpoonup N$ is partial and its graph is acyclic: nodes form a containment forest, and each containment link carries a slot label;
- $sc : E \to N \uplus \{\top\}$ gives each hyperedge the scope in which it is declared;
- $edge : I \to E$, $end : I \to X$, $role : I \to R$ and $ord : I \to \mathbb{N}$ say that incidence i attaches element $end(i)$ to hyperedge $edge(i)$ in role $role(i)$ at position $ord(i)$, and $(edge, role, ord)$ is injective: no two incidences of one hyperedge occupy the same slot;
- $kind : X \to K$ assigns every element a kind and $attr$ assigns typed attribute values.

Ownership, containment and scope together form one forest over X: ports hang under their owners, nodes under their parents, hyperedges under their scopes. This forest is the analogue of the place graph of a bigraph [247]. The

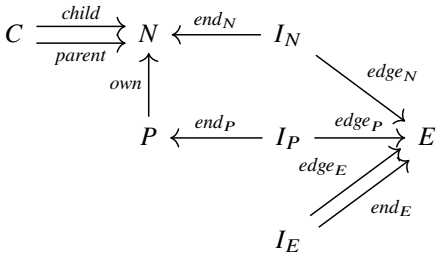


Figure 6: The schema $\mathcal{S}_{\mathsf{RG}}$. A Reaktor graph is a functor from this category to Set, with typed attributes (kinds, facets, roles, ordinals, contracts) attached to N, P, E and the incidence objects. In the kernel each object is a table and each arrow a column of foreign keys (Figure 19).

incidences form its link structure. Unlike bigraph links, they are typed, role-labelled and ordered. An incidence may attach a hyperedge to any element, so an assertion can be *about* a hyperedge, as in ubergraphs [189]. The signature (Definition 2) restricts executable hyperedges to ports. A hyperedge with k participants costs k incidences and keeps its identity. A clique expansion would cost $O(k^2)$ binary edges and lose both the identity and the roles (Table 3).

As attributed C-sets. Let $\mathcal{S}_{\mathsf{RG}}$ be the finitely presented category with objects N, P, E, I_N, I_P, I_E and C and generating arrows $own : P \to N$; $edge_s : I_s \to E$ and $end_s : I_s \to s$ for each sort $s \in \{N, P, E\}$; $child, parent : C \to N$; and attribute arrows into value types. A Reaktor graph is an attributed C-set, i.e. a functor $\mathcal{S}_{\mathsf{RG}} \to \mathsf{Set}$ with typed attributes, and a morphism of Reaktor graphs is a natural transformation that preserves attributes [280]. The conditions a C-set cannot state (the forest condition, slot injectivity, sort-dependent typing) become well-formedness axioms (Section 6.5). This presentation is what makes the kernel data-oriented: each object is a table and each arrow a column of foreign keys (Figure 6 and Section 11).

Proposition 1 (Adhesivity)**.** *The category* $\mathsf{RG} = \mathsf{Set}^{\mathcal{S}_{RG}}$ *of structural Reaktor graphs is a presheaf topos, hence adhesive. For any type graph* T*, the category of* T*-typed graphs* $\mathsf{RG}/T \simeq \mathsf{Set}^{\int T}$ *is again a presheaf topos, hence adhesive.*

Every presheaf topos is adhesive [210], and a slice of a presheaf topos is equivalent to presheaves on the category of elements. The consequence is that double-pushout (DPO) rewriting of typed Reaktor graphs has the Local Church–Rosser, Parallelism and Concurrency theorems [102], which Section 9 uses for transactions and merges. Attribute values in rules are treated as in [42].

5.2 Signatures and typing

Definition 2 (Signature)**.** A *signature* Σ consists of

- node kinds K_N, each with a set of *facets* and a port signature (required and optional port kinds);
- *families* K_E, each with a class (*connection* or *assertion*), a role set R_k, multiplicities $\mu_k : R_k \to \mathbb{N} \times (\mathbb{N} \cup \{\infty\})$, a polarity $\pi_k : R_k \to \{\mathsf{offer}, \mathsf{require}, \mathsf{neutral}\}$, admissible participant sorts and layers per role, a protocol and a lattice Q_k of quality-of-service levels;
- port kinds $K_P \subseteq \{(k, r) \mid k \in K_E,\ r \in R_k\}$: a port kind fixes the family and role in which the port participates.

The *type graph* $T(\Sigma)$ is the Reaktor graph with one node per node kind, one port per port kind (owned by every node kind whose signature lists it), one hyperedge per family and one incidence per admissible (family, role, port kind) triple.

A *typed* Reaktor graph is a graph G with a morphism $t : G \to T(\Sigma)$, so every element is mapped to its kind and every incidence to its role slot. This is the standard typed-graph construction of algebraic graph transformation [102]. Two consequences follow. First, a hyperedge's family and every participant's role are determined by the kinds of the ports involved: in Reaktor's design corpus, the family follows from the port profile in 95% of the connections whose ports imply one (Section 20.1). RG removes that redundancy rather than recording it twice. Second, the signature is open. A new domain kind is a new object of $T(\Sigma)$ (a new table), in the way a Kubernetes custom resource or an MLIR dialect [218] extends a fixed core without changing the kernel.

5.3 Layers

Definition 3 (Layers)**.** Let $\mathcal{L} = \{\mathsf{Src}, \mathsf{Def}, \mathsf{Dep}, \mathsf{Run}, \mathsf{Obs}, \mathsf{Ass}\}$: source, definition, deployment, runtime, observation and assurance. A *layered* signature assigns each node kind a layer $\lambda : K_N \to \mathcal{L}$. Connection families are *intra-layer*: all participants of a connection lie in one layer. Assertion families declare the admissible layer of each role and may cross layers. For $\ell \in \mathcal{L}$, let $\int T_\ell \hookrightarrow \int T$ be the full subcategory on the kinds and families of layer ℓ. The *projection* onto ℓ is the restriction functor $\pi_\ell = \Delta_\iota : \mathsf{Set}^{\int T} \to \mathsf{Set}^{\int T_\ell}$.

Layers answer different questions about one system. The *definition* layer says what is intended at a revision. The *source* layer says which artefacts implement it. The *deployment* layer says where it may run and on which resources. The

Representation	collective relation	typed ports and roles	nesting	cross-layer facts	rewriting theory	compositional semantics	size of a k-party relation
Directed/property graph	–	labels	–	labels	DPO	–	$O(k^2)$, identity lost
Tree (element tree, slot table)	–	–	✓	–	–	–	not expressible
Port graph [108, 133]	–	✓	✓	–	strategic	–	$O(k^2)$
Bigraph [247]	✓	arity only	✓	–	reactive	✓	$O(k)$
Open hypergraph [35]	✓	✓	–	–	DPO	✓	$O(k)$
Current runtime	–	partial	✓	–	–	–	not expressible
RG (this paper)	✓	✓	✓	✓	DPO	✓	$O(k)$, identity kept

Table 3: Why a typed port hypergraph. The bipartite incidence graph (the Levi graph of a hypergraph [30]) is not a competitor: it is how RG is stored.

runtime layer says what is running now, with which epochs. The *observation* layer records what happened. The *assurance* layer says what is claimed and on what evidence. Reaktor's design documents keep these statuses apart [293]. In RG they are fibres of one typed graph, and cross-layer facts are explicit assertions (Realizes, Places, Observes, Attests, Derives). In network-science terms [197, 323], RG is a heterogeneous, multiplex, multilayer hypergraph. Unlike the node-layer couplings of [197], a cross-layer relation in RG is a typed assertion with its own identity.

Proposition 2 (Views commute with patches)**.** *Restriction functors Δ preserve all limits and colimits. Hence the image under any layer projection, or any view obtained by restriction along a map of schemas, of a DPO derivation $G \Leftarrow D \Rightarrow H$ is a DPO derivation of the projected rule: the view of a patched graph is the patched view.*

Restriction has both a left and a right adjoint (left and right Kan extension), so it preserves pushouts and monomorphisms. This is the formal content of the design rule that a projection must never become a second source of truth. A view is a functor of the graph, not a copy, and the functorial data migrations Σ, Δ and Π [314] give a calculus for defining views and imports.

5.4 Identity, revision and activation

Definition 4 (Identities)**.** Every element x carries (i) a stable identifier $id(x)$, declared in source or minted at creation, typed by sort, never reused, and preserved by every patch that preserves x; (ii) a content digest $h(x)$, computed over the containment forest as a Merkle hash of x's kind, attributes, the sorted digests of its children and its sorted incidence tuples. The revision digest is $h(G) = h(\top)$. A contract c additionally has a fingerprint $fp(c)$ over its normalised structural description. An *activation* is identified by (definition identifier, place, key, epoch), where the epoch is issued by the authority that owns the activation and increases monotonically.

Four rules follow from the design corpus and from the current runtime's failures. Identifiers are never derived from labels, class names, memory addresses or construction order. A source-owned element gets its identifier from its declared name within its scope, and renaming it requires an alias, recorded as a Derives assertion. Equal fingerprints mean structurally identical contracts. Compatibility between different contracts is a separate, directional relation $\sqsubseteq$, so a newer server can serve an older client. Digests are canonical: permuting declarations does not change $h(G)$, and layout never enters it. Merkle digests make every scope a content-addressed unit, as in Nix and Unison [60, 98]. This is what lets a pushed region name exactly what it replaces.

6 The type system

The standard signature Σ_{std} is the vocabulary every Reaktor runtime understands. Four rules shaped it.

(T1) **Factor, do not enumerate.** A port type is a product of independent dimensions, a node kind is a set of facets, and two hyperedge families are separate only when their operational laws differ. The 2026 catalogue named 40 node kinds, 24 port profiles and 36 hyperedge kinds [291, 292]. Many of these were combinations of independent dimensions, and the same concept often appeared as a node, a port and a relation (Appendix B).

(T2) **Protocol is orthogonal to realization.** A family fixes *what* an interaction means. A binding fixes *how* it is carried: an in-process reference, an FFI trunk, inter-process messaging, RPC, a broker, a durable workflow engine or a peer mesh. Placement chooses the binding, which is admissible only if it refines the family's protocol at the required quality of service. "rpc" and "ffi" are therefore realizations, not port kinds. This is Service Weaver's split between logical components and deployment [143], made typed. It also keeps the Reaktor rule that location may be abstracted but latency and partial failure may not.

(T3) **Derived kinds are definitional extensions.** A screen, an actor, a workflow or a topic is a name for a pattern of base kinds, facets and families. Adding the name adds no semantics, so the kernel never depends on it.

(T4) **Facets, not subclasses.** Node kinds are archetypes of facets, as in entity–component systems [130, 157]. Unknown optional facets round-trip intact; unknown *required* semantics block activation.

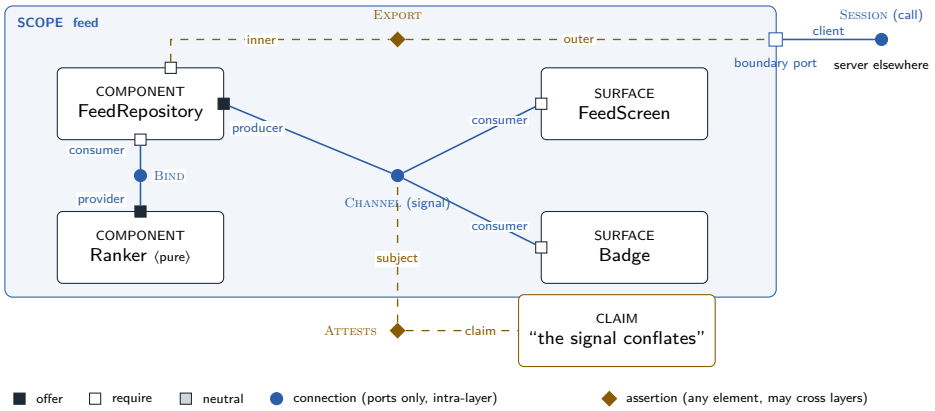


Figure 7: Anatomy of a Reaktor graph. Ports are owned by nodes and drawn on their borders; polarity follows from the port kind. A hyperedge is an element whose incidences carry roles: one CHANNEL joins a producer and two consumers without a clique of binary edges. The only incidence that leaves the scope passes through a boundary port, related to an inner port by EXPORT. The scope is therefore an open graph that can be mounted wherever a compatible SESSION server exists. The ATTESTS assertion takes a hyperedge as its subject.

6.1 Node kinds

Table 5 lists the fourteen base kinds, and Figure 2 uses all of them. Behaviour lives in Component. Its *profile* facet selects among pure, effectful, exclusive (actor), stream, durable, native and human, and each profile carries laws, not just a label. An exclusive component processes one message at a time from its mailbox and owns its state. A durable component's progress is persisted at a named store, so a coroutine stack is never treated as a checkpoint. Scope is the unit of composition, isolation, supervision and update. Route is an addressable navigation state. Being scope-like, routes nest, and a surface mounted at a route is its child. Store is a consistency domain, a named authority that commits state. Dataset is a logical relation whose schema is a Contract. A table is therefore part of the graph, but a row never is. A dataset's contract may also declare invariants (row checks, uniqueness, acyclicity, referential integrity, sums over rows), which its store enforces inside its transactions (Section 9).

Anatomy of a node. Every node has five parts:

- a *stable identifier*;
- a *kind*, which fixes its layer and its required facets;
- an *owner* in the containment forest: its scope, or, for a port, its node;
- a set of *facets*, typed attribute groups that compose like the components of an entity–component system;
- the *ports* it owns.

Identifiers derive from declaration paths: the scopes from the root, the local name and the kind, hashed to 128 bits. They survive rebuilds, reordering and moves between files. Elements that placement or the runtime create (realizations, activations, observations) derive theirs from the element they realize, a key and an epoch. A restarted activation therefore has a new identifier that still traces back to its definition.

Facets. A facet is a typed record with required and optional fields and its own rules. A kind declares the facets its nodes must carry, and a node may carry more. No two facets may define the same field, so facets compose without conflicts, and a tool that does not know a facet passes it through unchanged (T4). Labels, tenancy, retention and supervision are facets. That is why the additions of Section 6.4 needed no new kind.

Profiles. The profile facet of a Component carries laws, not just a label. Table 4 states what the runtime guarantees for each profile, what the checker verifies before activation, and how the reference stack realizes it.

Derived kinds. A derived kind is a definitional extension (T3): a name for a base kind with required facets, ports and constraints. For example,

$$
\begin{aligned}
\text{actor} &\triangleq \text{Component}\langle\text{exclusive}\rangle \text{ with one (CHANNEL, consumer) port on a mailbox,}\\
\text{screen} &\triangleq \text{Surface mounted at a Route,}\\
\text{topic} &\triangleq \text{Resource that REALIZES a fan-out CHANNEL,}\\
\text{workflow} &\triangleq \text{Component}\langle\text{durable}\rangle \text{ whose body is a TRANSITION net,}\\
\text{schedule} &\triangleq \text{Component that binds its place's clock and produces a timer CHANNEL.}
\end{aligned}
$$

The kernel stores only base kinds and facets. A derived kind is a query over them, so a tool that does not know the name still handles the element correctly.

Profile	Law the runtime guarantees	Checked before activation	Typical realization
pure	No effects; the same inputs give the same outputs, so results may be cached or replayed	No Access writes and no effectful sessions; captures only values	Inline call; memoised
effectful	Effects only through declared Access and Session ports; its effect is the join of that footprint	Footprint within the ceiling (W10) and covered by grants (A5)	Any compute place
exclusive	One message at a time from its mailbox; owns its state; one live instance per key	A single mailbox consumer; no shared mutable state	Durable Object; actor host
stream	Ordered inputs with windows, watermarks and back-pressure	Ordered input channels; time attributes present	Coroutine flow; stream processor
durable	Progress persisted at a named store; resumes by replaying committed outcomes	Steps are code entries; no captured stacks	Workflows; Temporal; alarms
native	Runs in a native runtime behind an FFI trunk; ownership enforced at the boundary	Ownership modes declared; a borrow stays inside its owner's scope (W9)	C++ via JNI, cinterop, JSI or Wasm
human	A person acts through a surface; a step may wait for days	A deadline and an escalation on every human session	Approval surface with a durable wait

Table 4: The profiles of Component and their laws.

6.2 Ports

Definition 5 (Port type)**.** A port type is $\tau = (k, r, c, \omega, q)$: a family k, a role $r \in R_k$, a contract fingerprint c, an ownership mode $\omega \in \{\mathsf{value}, \mathsf{move}, \mathsf{borrow}, \mathsf{share}\}$ and a quality-of-service requirement $q \in Q_k$. Polarity is derived, $\pi(\tau) = \pi_k(r)$; it states which side offers a contract, not which way each message travels.

The ownership modes carry the obligations that FFI needs [349]. A move port transfers a resource that must be accepted exactly once. A borrow port lends a view whose lifetime nests inside the owner's scope and which cannot leave its process. A share port grants a read lease. A camera frame handed to a native encoder, a model's weights shared between workers and a FlexBuffers view decoded in place are three modes over the same families. The quality-of-service lattice Q_k is a product of chains (synchrony, delivery, ordering, retention, durability, deadline, capacity), ordered by strength, and the requirement of a hyperedge is the join of its ports' requirements. The terms in common use map onto port types as follows. A *provider* port is (Bind, provider) or (Session, server). A *receiver* is (Channel, consumer). A *broadcast* port is a (Channel, producer) on a fan-out channel. An *rpc* port is a Session call realized over a network, and an *ffi* port is any port realized by an FFI *trunk*, a typed connection between two runtimes in one process, such as JSI between a Hermes engine and its host. Its ownership is usually move or borrow.

Polarity. Polarity says which side offers the contract. The provider of a Bind, the server of a Session, the producer of a Channel and the pre-state of a Transition offer; the others require. An interceptor does both: it requires what it forwards and offers what it returns.

Contracts and compatibility. A contract is a schema or protocol in a canonical form, and its fingerprint is a digest of that form. Compatibility $c \sqsubseteq c'$ means that c may be used where c' is required. It is computed, never declared: a version number is a label, not evidence.

- *Records.* An offered record may add optional fields, but it may not drop or retype a field that the requirement reads.
- *Sessions.* The server must accept every request the client may send, so requests are contravariant. The client must accept every outcome the server may return, so outcomes are covariant.
- *Channels.* Every consumer must accept every producer's messages, so the channel's message type is the join of what producers send, and it must lie below what each consumer accepts.

Equal fingerprints decide compatibility in constant time. Otherwise the kernel computes the relation once per pair of fingerprints and caches it.

Quality of service. Table 6 lists the dimensions of Q_k. Each is a chain ordered by strength. Capacity and overflow are attributes rather than chains, because no overflow policy is stronger than another in every use. A binding qualifies for a hyperedge when its guarantee dominates the join of its ports' requirements in every dimension (A2).

6.3 Families and hyperedges

Table 7 gives the four connection and ten assertion families. Connections have operational semantics (Section 7), so they participate in execution. Assertions are facts used by analysis, planning and admission; they never execute. This is the corpus's own distinction between executable *connections*, semantic *relations* and local *facets* [287], now made precise.

Four choices in Table 7 need justification. First, Bind is local by construction. A reference between processes would pretend that a network call cannot fail. Within one process, runtimes are places nested in the process's place, and a Bind between them is realized by an FFI trunk that checks ownership. Remote dependencies are therefore Sessions, and the type checker can see the difference. Second, Session subsumes request–reply, streams and duplex sessions because each is a global type: a call is client $\to$ server: *Req*. server $\to$ client: *Outcome*, where *Outcome* distinguishes a

Base kind	Layer	Characteristic facets and attributes	Derived kinds (definitional)
Scope	Def	boundary ports; supervision (one-for-one, all-for-one, escalate); isolation (none, process, memory, principal); update unit; edit owner (source or graph); opacity	module, feature, region, factory, external system
Component	Def	profile (pure, effectful, exclusive, stream, durable, native, human); code entry; turn policy; effect ceiling	interactor, repository, service, actor, worker, workflow, step, policy, adapter, model, controller, check
Surface	Def	named slots; form-factor guards; renderer profile	screen, container, input source, form
Route	Def	path pattern; argument contract; history (shallow, deep)	deep link, modal, tab
Store	Def	consistency profile (transactional, serialized, consensus, convergent, append-only); isolation; durability	database, key–value store, blob store, actor store, ledger
Dataset	Def	schema contract; representation (authoritative, derived, replica); data family (relational, document, graph, vector, search, log); invariants; retention	table, view, index, materialization
Contract	Def	fingerprint; compatibility relation; interaction protocol	schema, service contract, capability, event type
Principal	Def	type (person, role, service account, agent, organization)	agent, tenant
Artifact	Src	digest; format; target; authored or built	module, bundle, binary, manifest, patch, plan, release, migration
Place	Dep	runtime (ART, JVM, Native, V8, Hermes, CPython, database); trust tier; region; capacity; offered capability ports	device, browser tab, isolate, process, host, pool, region
Resource	Dep	provider; locator; configuration	bucket, topic, queue, database instance, Durable Object namespace
Activation	Run	key; epoch; lifecycle state; parent activation	run, attempt, session instance, mount
Observation	Obs	type (span, event, metric, log, receipt); time; coverage; normally virtual	trace, receipt, sample
Claim	Ass	proposition; assumptions; bounds; status per dimension	invariant, finding, approval, objective

Table 5: The fourteen base node kinds of Σ_{std}. Observations are usually *virtual*: they keep stable identities and types but live in their own stores (for example ClickHouse) and are materialised on demand.

Family	Dimensions of Q_k, each ordered from weak to strong
BIND	resolution: lazy < eager; selection is an attribute (exclusive, ordered fallback, all, keyed)
SESSION	synchrony: asynchronous < synchronous; deadline: none < seconds < milliseconds; effect identity: none < keyed < keyed with request digest; acceptance: volatile < durable; retry, hedging and batching policies are attributes
CHANNEL	delivery: best effort < at least once < effectively once; order: none < per key < total; retention: transient < bounded < durable; replay: none < from offset; capacity and overflow are attributes
TRANSITION	durability: volatile < durable; firing: at most once < exactly once per token; the join policy is an attribute

Table 6: Quality-of-service dimensions per connection family.

domain rejection, a failure and an unknown effect. Third, CHANNEL is separate from SESSION because its participants are dynamic and decoupled by a buffer. A topic does not know its subscribers, whereas a session's roles are fixed when it opens. Fourth, navigation, workflow steps and sagas share TRANSITION: each moves control between states under a guard, and they differ in their attributes. A bottom-navigation container, whose tabs keep separate back stacks, is a statechart: orthogonal regions with history states (Figure 13).

Anatomy of a hyperedge. A hyperedge has an identifier, a family, the scope that declares it, attributes, and a set of incidences. An incidence is (e, x, r, i): the hyperedge, the participant, the role and a position. For a connection the participant is a port; for an assertion it may be any element. The position orders participants within one role, where order matters: the fallback order of a BIND selector, or the order in which interceptors see a call. Otherwise it is zero. In the 140 published design graphs, 542 of 648 relations are binary, 88 ternary and 18 have four participants. The 16% that are not binary include sessions with interceptors, fan-out channels, joins and grants. Encoded as binary edges they would lose their identity and their roles (Figure 3).

Typing rules. The judgments of Section 6.5 are generated by a few rules. Figure 9 gives the core ones. Here $\tau(p)$ is the type of port p, $\mu_k(r)$ the multiplicity of role r, π polarity, and $q(e)$ the requirement of hyperedge e. (Inc) types one incidence. (Conn) types a connection: roles within their multiplicities, every offer compatible with every requirement, and a requirement that is the join of its ports' requirements. (Sess) adds the global type. (Exp) types the leg of an open graph, and (Real) says when a binding may carry a connection.

A typed example. In the chat scope of Listing 7, the timeline is a CHANNEL hyperedge with one producer incidence (the actor's `timeline` port) and two consumer incidences (the two surfaces' `events` ports). Its requirement is the join of theirs: fan-out, latest value, at least once, ordered per room. A Durable Object's hibernatable WebSockets meet that requirement, while a competing-consumer queue does not, because it delivers each message to one consumer. The call to `MessagingApi` is a SESSION with a client, a server and one interceptor at position 0. (Sess) checks the three local types

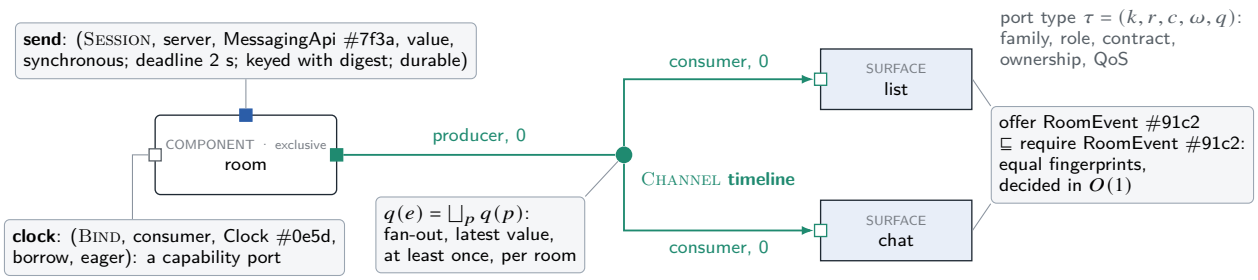


Figure 8: Typed ports and one hyperedge from the chat scope. Each port carries a full type τ. The CHANNEL hyperedge joins one producer and two consumers; its requirement $q(e)$ is the join of theirs, and every offered contract must be compatible with every required one. A filled square offers its contract and a hollow one requires it.

$$\frac{kind(e) = k \qquad r \in R_k \qquad \tau(p) = (k, r, c, \omega, q)}{\vdash (e, p, r, i)\ \mathsf{ok}}\ \textsc{Inc}$$

$$\frac{\forall r \in R_k.\ \big|\{(e, p, r, i)\}\big| \in \mu_k(r) \qquad \forall p \in \mathit{off}(e),\ p' \in \mathit{req}(e).\ c(p) \sqsubseteq c(p')}{\vdash e : k\langle q(e)\rangle \quad \text{where } q(e) = \bigsqcup_{p \in e} q(p)}\ \textsc{Conn}$$

$$\frac{\vdash e : \textsc{Session}\langle q\rangle \qquad G \text{ well formed} \qquad \forall r.\ T(p_r) = G \upharpoonright r}{\vdash e : \textsc{Session}\langle G, q\rangle}\ \textsc{Sess}$$

$$\frac{p \in \mathit{ports}(n) \qquad n \in \mathit{subtree}(s) \qquad p' \in \partial s \qquad \tau(p) \sqsubseteq \tau(p')}{\vdash \textsc{Export}(p, p')\ \mathsf{ok}}\ \textsc{Exp}$$

$$\frac{\vdash e : k\langle q\rangle \qquad b \text{ refines the protocol of } k \qquad \mathit{guar}(b) \sqsupseteq q}{\vdash b \models e}\ \textsc{Real}$$

Figure 9: Core typing rules for incidences, connections, sessions, exports and realizations. Here $\mathit{off}(e)$ and $\mathit{req}(e)$ are the offering and requiring ports of e, ∂s the boundary ports of scope s, and $G \upharpoonright r$ the projection of the global type G onto role r.

against the call's global type, and (Real) admits a service binding that carries the deadline and the idempotency key.

Services. A `Service` of the current runtime becomes a Component whose SESSION server ports *are* its operations, typed by one Contract. The client's port references the same contract fingerprint, so client and server cannot drift. The HTTP method and route, a Cloudflare service binding or a Durable Object stub are attributes of the binding chosen at placement, derived from the contract rather than written twice. A worker's graph and a phone's graph are no longer joined after the fact by matching contract names: both are fibres of one graph, and the operation that crosses between them is one hyperedge.

Networking. Networking needs no family of its own. A service binding, an HTTP route, a WebSocket or a database connection is a connection in the *deployment* layer, between ports of Places and Resources, with the network's latency, bandwidth and trust as attributes. The definition-layer connection that uses it is related to it by REALIZES. Condition (A2) below is the check that such a realization exists and is qualified.

6.4 Effects, labels, tenancy and time

Four further rules came from mapping a production platform onto the signature (Section 20.2). None adds a kind or a family.

Effects. The signature fixes one effect lattice:

$$\text{read} < \text{live read} < \text{non-production write} < \text{reversible production write} < \text{credential write} < \text{data migration} < \text{destructive} < \text{unknown remote effect}.$$

An operation's effect is the join of the effects in its declared ACCESS and SESSION footprint, never something inferred from its name. Effect ceilings (W10), and the effect class above which a plan needs approval (Figure 15), are stated in this lattice. It replaces the separate effect vocabularies that tools otherwise invent.

Family	Roles (multiplicity)	Meaning and principal attributes	Derived forms
Connections (ports only; intra-layer; executable)			
Bind	provider (1..n), consumer (0..n), selector (0..1)	A co-located reference or value, resolved at activation and invoked as a direct call. Selection is exclusive, ordered fallback, all, or keyed. It never leaves a process.	DI binding, capability binding, fallback ladder, multibinding, source import
Session	roles of a global type G; interceptor (0..n)	A multi-message protocol whose participants follow the projections of G [173]. An interceptor sits on the path between two roles: it receives each message first and passes, rejects or annotates it, so its local type is fixed by those two roles' projections. Attributes: synchrony, deadline, idempotency key, durability of acceptance.	call, stream (credit-based), duplex, durable call, multiparty session
Channel	producer (0..n), consumer (0..n)	One-way messages through a buffer. Distribution: fan-out or competing consumers. Also retention, delivery, ordering, scope, capacity, overflow and dead-letter target.	event, signal (state), topic, queue, mailbox, timer (fed by a schedule)
Transition	pre (1..n), post (1..m), trigger (0..1)	Transfer of control with Petri-net firing [258]. Join: all, any, first, quorum k or barrier. Also guard, mode (push, replace, pop, return) and durability. Hierarchy comes from containment, as in statecharts [163].	navigation, workflow step, fork, join, compensation
Assertions (any element; may cross layers; never executed)			
Export	inner (1), outer (1)	Aliases a scope's boundary port to an inner port: the leg of an open graph.	route export
Access	accessor (1), dataset (1..n)	Effect footprint: read, write or subscribe, and the atomicity domain.	transaction scope
References	from (1), to (1)	Reference between elements (dataset fields, artefacts, principals), with a kind, cardinality and integrity rule.	foreign key, citation
Replicates	authority (0..1), replica (1..n)	Replication with direction, checkpoints and conflict law.	cache, sync view
Grant	grantor (1), grantee (1), resource (1..n), policy (0..1)	Authority to act on resources, with actions, expiry, delegation depth, rate and budget.	delegation
Places	subject (1..n), target (0..1), peer (0..n)	Desired placement: on, near or apart, or partitioned by key.	affinity, shard map
Realizes	spec (1), impl (1..n)	Implementation, binding or instance, with epoch and key.	instance, cohort member
Derives	input (0..n), transform (0..1), output (1..n)	Provenance and lineage under a revision cut [255].	build, migration, projection
Observes	observer (1), subject (1..n)	An observation of elements, with coverage and sampling.	span, metric
Attests	claim (1), subject (0..n), evidence (0..n), verifier (0..n)	Claim-indexed evidence with method, assumptions and freshness.	finding, approval

Table 7: Families of Σ_{std}. With the containment forest, they cover the 36 relation kinds of the 2026 catalogue; three of those kinds (temporal join, clock alignment, tensor operation) become stream components rather than relations (Appendix B).

Labels. A Grant says who may act; it does not say where data may flow. RG therefore adds a *label* facet on dataset fields, contracts, channels and observations [91, 259]. Labels are drawn from a lattice of confidentiality levels (public ≤ internal ≤ personal ≤ sensitive ≤ secret), refined by purpose tags such as model use or analytics.

- *Propagation.* Labels propagate along connections and Derives by a monotone traversal, so derived data inherits the strictest label of its inputs.
- *Declassification.* A declassifier is a component holding a grant to lower a label for a stated purpose. It is the only legal way across.
- *Checking.* Rule W11 checks statically known paths, and executors check the rest at run time.

Privacy rules, restrictions on model use and the containment of untrusted input to agents thereby become one mechanism. Figure 10 shows both orders.

Tenancy. A multi-tenant store, place or activation carries a tenant partition facet. Rule A6 makes isolation a checked property of the graph rather than a convention of each query.

Time. Time is a capability of a place. A *schedule* is a derived kind of Component: a Channel producer that binds the clock capability of its place, so the channel stays in the definition layer. It has calendar semantics (a cron or recurrence rule and a time zone), a misfire policy (skip, fire once, or catch up) and jitter. Durable timers inside durable transitions are its other form. Scheduled work is therefore a connection in the graph rather than a synthetic request. Listing 1 declares all four in one scope.

6.5 Well-formedness

Well-formedness has a static part, which every revision must satisfy, and an activation part, which depends on placement.

Static ($\mathsf{WF}_{\mathsf{static}}$).

(W1) *Forest.* *own* is total, and $own \cup par \cup sc$ is acyclic.
(W2) *Kinding.* $t : G \to T(\Sigma)$ is a morphism; every node has the ports its kind requires, or a *typed hole* in their place [271].
(W3) *Roles.* For each hyperedge e of family k and each $r \in R_k$, the number of incidences of e in role r lies in $\mu_k(r)$.

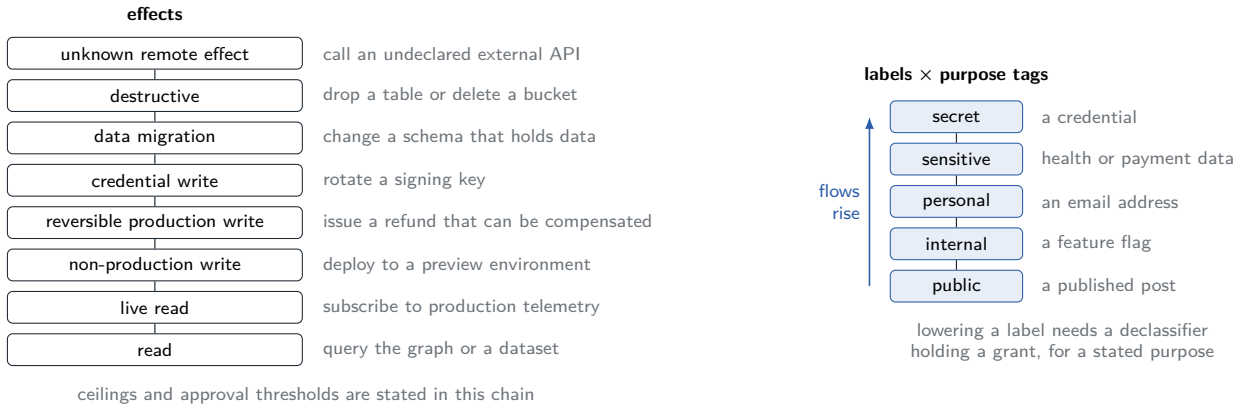


Figure 10: The two orders that every operation and datum carries (Section 6.4), each with an example per level. An operation's effect is the join of its footprint; a datum's label is the join of its sources, refined by purpose tags such as model use or analytics.

```
val Ledger = scope("ledger", owner = Source, tenancy = PerTenant) {
    val bank = require(Session.call<BankApi>())
    val db = store("ledger-db", Transactional) {
        dataset<Entry>("entries") {
            key(Entry::id)
            label(Entry::counterparty, Personal)
            label(Entry::memo, Sensitive, purpose = Audit)
            invariant("balanced") { groupBy(Entry::transferId).sum(Entry::amount) eq 0 }
            retention(years = 8)
        }
    }
    val nightly = schedule("reconcile", cron = "0 2 * * *", zone = UTC, misfire = CatchUp)
    component("reconciler", ::Reconciler, profile = Durable, ceiling = ReversibleProductionWrite) {
        consumes(nightly); reads(db["entries"]); calls(bank)
    }
    grant(to = role("finance"), on = db["entries"], actions = setOf(Read), rate = perMinute(120))
}
```

Listing 1: Labels, invariants, tenancy, schedules and quotas in one scope (proposed API).

(W4) *Nets.* A port is incident to at most one connection, in the family and role fixed by its kind. A port is a pin, a connection is a net, and fan-out is expressed by the net, not by the pin.

(W5) *Contracts.* In every data-carrying connection, each offered contract is compatible with each required one, $c_{\mathsf{offer}} \sqsubseteq c_{\mathsf{require}}$, and the session participants' local types are projections of one well-formed global type; an interceptor's local type is fixed by the two roles it sits between. The outcome type of every SESSION whose effect is above read-only includes UNKNOWN.

(W6) *Layers.* Connections are intra-layer; each assertion role is filled from its declared layers.

(W7) *Boundaries.* Every participant of a hyperedge declared in scope s lies in the subtree of s, except through boundary ports related by EXPORT.

(W8) *Cycles where they matter.* Eager BIND edges are acyclic; a cycle of pure value dependencies needs a declared delay; other cycles (loops, recursion, feedback) are legal.

(W9) *Ownership.* A move port is consumed by exactly one participant, and a borrow incidence stays within its owner's scope.

(W10) *Effect ceiling.* A scope's ACCESS footprint and outgoing sessions are within its declared ceiling. This bound is what a pushed bundle may touch.

(W11) *Flow.* A connection carries values only to ports whose clearance dominates the values' labels, unless a declassifier on the path holds a grant for that purpose (Section 6.4).

Activation ($\mathsf{WF}_{\mathsf{act}}$, for a placement p).

(A1) *Locality.* Every BIND joins ports of components in one process: on the same place, or on runtime places nested in the process and joined by an FFI trunk.

(A2) *Realizability.* Every connection has a binding whose guarantees meet the hyperedge's requirement and that refines the family protocol on the runtimes involved. A synchronous session in a browser cannot be silently realized as asynchronous RPC.

(A3) *Availability.* Every capability port, a (BIND, consumer) port whose provider must be an offer of the component's

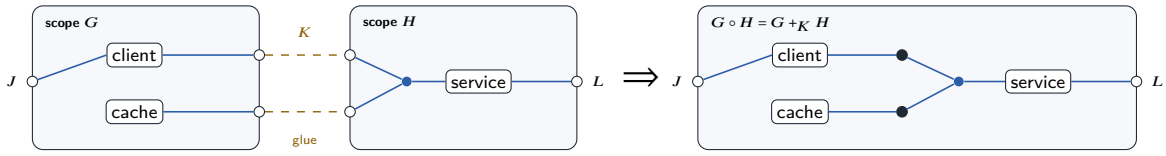


Figure 11: Composing open graphs. Each scope is a cospan of its boundaries ($J \to G \leftarrow K$ and $K \to H \leftarrow L$). Composition glues the shared boundary K by pushout. The glued ports become internal and their nets join, here into one three-party hyperedge. Theorem 3 makes behaviour respect this gluing, so a scope can be mounted at any boundary that matches (Corollary 4).

place, is offered by that place at or above its required support (Section 6.6).

(A4) *Closure.* There are no typed holes, and every required port is bound.

(A5) *Authority.* Every effectful connection is covered by a Grant. The default is deny, and reads count as effects. A grant to approve a proposal never covers the proposal's author.

(A6) *Tenancy.* Every Access to a multi-tenant dataset carries the activation's tenant, and every flow across tenants is covered by an explicit Grant.

These are the judgments of the stable-foundations audit (*WFStatic*, *WFActivation*) with their content made explicit [295]. A draft may violate (W2) and (A4) and still be saved, shown and analysed, but it cannot be activated.

6.6 Support lattices and resolution

The design corpus contains four ordered vocabularies that answer the same question: can this offer meet this requirement, and how well? They are capability tiers (unavailable, fallback, basic, standard, full); data bindings (unsupported, emulated by a reviewed plan, supported); renderer support; and FFI schema matching (incompatible, keyed compatibility, positional). RG treats them uniformly.

Definition 6 (Support)**.** A *support lattice* S is a finite bounded lattice. Requirements $\mathcal{R}$ and offers O are related by a support function $s : O \times \mathcal{R} \to S$ that is monotone in the offer and antitone in the requirement. A requirement r carries a threshold $\theta_r \in S$. The eligible offers are $\mathrm{Elig}(r) = \{o \mid s(o, r) \geq \theta_r\}$, and resolution picks a $\leq$-maximal element of $\mathrm{Elig}(r)$, breaking ties by preference.

Elig and its dual, the set of requirements that every offer in a set meets, form a Galois connection between sets of offers and sets of requirements [137]. The resulting closure is what placement computes: the largest set of places that satisfies every hard requirement. Preferences such as cost, latency or energy rank only within that set, so "no silent downgrade" becomes a property of the threshold rather than a review comment. The support of a composite is the meet of its parts' supports, which makes eligibility monotone and computable by the traversals of Section 8. Authority is *not* a support level. Availability says what a place can do; a Grant says who may ask it to.

7 Composition and behaviour

7.1 Open graphs

A scope that interacts with its context only through boundary ports is an *open graph*. To give open graphs an algebra, we encode the connection fragment of a Reaktor graph as a hypergraph in the sense of string-diagram rewriting [35]. Each port becomes a vertex. Each node becomes a box labelled by its kind, attached to its ports' vertices. Each connection becomes a box labelled by its family, attached to its participants' vertices in role order. The net discipline (W4) gives every vertex at most one connection box. A boundary port is a vertex left open.

Definition 7 (Open Reaktor graph)**.** Let Σ° be the monoidal signature whose generators are node kinds and family instances, typed by their ordered port types. An *open Reaktor graph* $G : J \to K$ is a cospan $J \to \mathcal{E}(G) \leftarrow K$ of Σ°-hypergraphs, where $\mathcal{E}(G)$ encodes G and J, K are discrete hypergraphs of open, typed ports. Composition along K is pushout, which plugs open ports into the waiting nets.

Theorem 3 (Compositional semantics)**.** *Assign to every generator of Σ° a behaviour: a span of labelled transition systems from the generator's transition system to the alphabets of its ports. Then:*

(a) the encoding of Appendix A sends well-formed open Reaktor graphs, up to isomorphism, injectively to morphisms of the free hypergraph category on Σ°, and mounting a scope at its boundary preserves well-formedness;

(b) there is a unique hypergraph functor $[\![-]\!]$ from that category to $\mathsf{Span}(\mathsf{Graph})$ extending the assignment, so that $[\![G \circ H]\!] = [\![G]\!] \circ [\![H]\!]$ (synchronise on shared ports, then hide) and $[\![G \otimes H]\!] = [\![G]\!] \otimes [\![H]\!]$;

(c) composition is monotone for trace refinement: if $[\![S']\!] \sqsubseteq_{\mathrm{tr}} [\![S]\!]$ then $[\![C \circ S']\!] \sqsubseteq_{\mathrm{tr}} [\![C \circ S]\!]$ for every context C.

Part (a) is the correspondence between discrete cospans of hypergraphs and free hypergraph categories [35], in its many-sorted form. Part (b) follows from freeness, since Span(C) is a hypergraph category for any finitely complete C [136]. Composition in Span(Graph) is the synchronised product of transition systems introduced as Span(Graph) [193]. Part (c) holds because a trace of a pullback is a pair of compatible traces (Appendix A).

Two corollaries carry the requirements R1 and R11.

Corollary 4 (Mount anywhere)**.** *If two contexts have equal behaviour at a boundary J, then mounting the same scope with boundary J in either produces equal behaviour. A self-contained scope therefore depends on its context only through J.*

Corollary 5 (Sound replacement)**.** *If a candidate region S' refines the boundary traces of the region S it replaces, then no context can observe a boundary trace with S' that it could not observe with S.*

Corollary 5 is the semantic core of code push. It does not by itself cover state migration, in-flight sessions or durable continuations; Section 9.5 adds those obligations. Three limits are deliberate. First, trace refinement captures safety. Liveness requires assume–guarantee reasoning with explicit fairness [2]. Second, asynchrony lives inside generators: a Channel is a buffer automaton, and composition stays synchronous on events. This avoids the Brock–Ackerman anomaly [38], which breaks relational semantics of nondeterministic dataflow over untimed histories. It also matches the timed-stream semantics of FOCUS [43]. Third, an opaque Kotlin body contributes its declared contract behaviour, an over-approximation. Its actual behaviour is checked by tests and, for selected protocols, by models (Section 15).

Scopes as operations. A scope with unfilled slots is an operation of the operad of wiring diagrams [315], and filling slots is operadic composition. Associativity and unit laws say that the order in which nested regions are mounted does not matter. This is the algebra behind the current runtime's nested graphs, made explicit so that it can be checked.

7.2 Operational semantics and its TLA$^+$ projection

The state of a running system is $\sigma = (\Gamma, A, H, Q, D, \mathcal{E})$. Here Γ is the set of admitted revisions, A the activations with their keys and epochs, H the local states of activations, Q the protocol state of each realized connection (buffers, calls in flight, credits), D the store states, and $\mathcal{E}$ the leases and epochs held by authorities. A step is labelled by a local computation, a send, delivery or acknowledgement on a connection, a commit at a store, a firing of a transition, an activation or deactivation, a crash or recovery, a revision publication, or an ownership handover.

Each family contributes a parameterised state machine. A Channel with fan-out, at-least-once delivery and per-key order is a per-consumer queue with Publish, Deliver, Ack and Redeliver. A Session call has Request, Accept, Commit(*effectId*), Reply, LoseReply, Timeout and Lookup. An idempotency key binds the digest of its request: a replay with the same key and a different digest is a rejection, not a second effect [219]. Because composition in Span(Graph) is a synchronised product, the TLA$^+$ reading is conjunction over shared variables, with hiding as temporal existential quantification [2]:

$$Spec(G) \;\triangleq\; \exists\, internal : \bigwedge_{n \in N} Spec_{\kappa(n)}(n) \;\wedge\; \bigwedge_{e \in E_{\text{conn}}} Spec_{\kappa(e)}(e) \;\wedge\; Env, \tag{1}$$

where $\kappa(x)$ is the kind of a node or the family of a connection, E_{conn} is the set of connection hyperedges, *internal* ranges over the variables of internal ports, and *Env* constrains the boundary ports. A binding is *qualified* for a family at a quality level when its implementation refines the family's specification under a refinement mapping [1]. For example, Cloudflare Queues refine a competing-consumer, at-least-once Channel, but not an ordered one; Google Cloud Pub/Sub with ordering keys refines a fan-out, at-least-once, per-key-ordered one. Condition (A2) consults these qualifications. Placement therefore never chooses a binding by name; it chooses one whose refinement has been established, by proof or by a qualification suite.

Execution profiles. The family fixes what composition means, and the component profile fixes what a component may do. A pure component computes a function. An effectful one acts sequentially through its ports. An exclusive one takes one message per turn from its mailbox. A stream component transforms sequences under demand. A durable one persists each step's committed outcome and recovers by replaying outcomes rather than stacks. A native one is opaque and bound by its contract, and a human one has unbounded latency and its own authority. Attaching a model of computation to a region and composing heterogeneous regions through typed boundaries follows Ptolemy [104]. In RG the composition rule is Theorem 3, and each profile's guarantees are explicit and are never inherited by a neighbour.

8 Traversals: one computational primitive

8.1 Definition

Definition 8 (Traversal)**.** A traversal is a tuple $\theta = (\sigma, V, \mu, \oplus, \upsilon, \beta)$. Here σ is a *selector* (the element kinds and the directed role pairs $(k, r \to r')$ to follow, including containment); V is a value domain; $\mu_{k,r\to r'} : V \to V$ are *message*

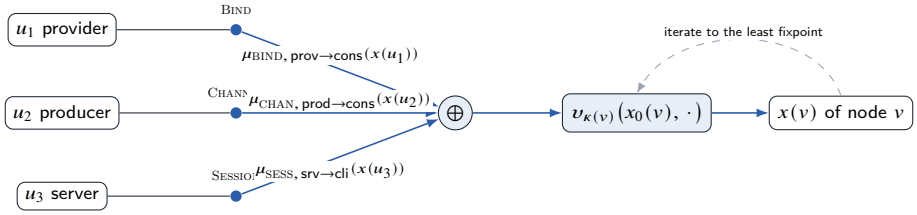


Figure 12: One step of a traversal (Equation (2)). Along each selected incidence, a message function chosen by family and role pair transforms the neighbour's value. Messages are aggregated by an associative, commutative $\oplus$ and combined with the node's own value by an update chosen by its kind. Iterating to the least fixpoint gives the result, in any order (Theorem 6).

functions; $\oplus$ is an associative, commutative *aggregation*; $\upsilon_k : V \times V \to V$ are *update* functions, one per node kind k; and β is a budget. On a graph G, θ denotes the least solution of

$$x(v) \;=\; \upsilon_{kind(v)}\Big(x_0(v),\; \bigoplus_{(u,\,k,\,r\to r')\,\in\,\sigma(v)} \mu_{k,r\to r'}\big(x(u)\big)\Big), \tag{2}$$

where $\sigma(v)$ enumerates the selected incidences arriving at v, when that least solution exists. Theorem 6 gives sufficient conditions for lattices; a path problem needs a closed semiring, so counting paths needs an acyclic graph or a bound. A traversal returns a value assignment (a *query*), a patch (a *rewrite*) or a patch to the runtime or deployment layer (a *plan*). It always also returns findings and a *coverage claim*: the revision, the layers and scopes visited, the elements the budget excluded, and the opaque boundaries it could not see.

What the current runtime calls a visitor is the pair (μ, υ) of a traversal. The traversal adds what those visitors left implicit: the selector, the algebra, the schedule and the budget. Equation (2) is one step of a graph network [29, 145], generalised from vertices and edges to typed incidences with roles. A Transformer layer is the special case of a complete graph with a learned, attention-weighted aggregation [344, 346]. The analogy is deliberate. One parallel primitive replaced the bespoke recurrent and convolutional architectures; here one typed primitive replaces the bespoke algorithms that the current runtime hand-wrote inside classes: wiring, shape extraction, deep-link search, telemetry installation and rendering. Because the representation is a set of incidence tables (Section 11), learned analyses can later read the same tensors.

8.2 Algebraic classes

The traversals of interest compute in four classes, each with its own guarantees. Rewrites and plans come from any class: they are traversals whose result is a patch.

(a) **Folds over the containment forest.** These are attribute grammars [205]. *Inherited* attributes flow down: the injection environment, theme, authority context and layout constraints. *Synthesized* attributes flow up: sizes, effect footprints, exported ports, readiness and digests. They are incrementally re-evaluable after an edit, the classic technique of language-based editors [297].

(b) **Path problems over a semiring** $(S, \oplus, \otimes, \bar{0}, \bar{1})$ [194, 250]. Boolean for reachability (unreachable routes, effect reachability, taint [298]); min-plus for the shortest route; max-plus for the worst-case latency of an acyclic call chain; max-min for capacity; counting for path multiplicities in acyclic graphs; and provenance polynomials [156] to explain *why* a finding holds.

(c) **Fixpoints over lattices**, i.e. Datalog with lattice values [4, 232]. These cover dependency resolution with ambiguity detection, capability eligibility, effect ceilings, contract inference and compatibility closures.

(d) **Solvers**: placement (integer programming), partitioning into regions [192] and layout. These are not monotone and are not least solutions of (2). A traversal of the first three classes builds the constraint system, an external solver solves it and returns a certificate, and traversals of the first three classes validate the result before admission.

Theorem 6 (Determinism and cost)**.** *Let V be a lattice of finite height h, $\oplus = \sqcup$, and let every μ and υ be monotone. Then (2) has a least solution. Every fair chaotic iteration from $\bot$ reaches it, whatever the order, parallelism or partition of the graph. A worklist schedule performs at most $h \cdot |X|$ updates and $O(h \cdot |I|)$ message evaluations.*

This is the asynchronous-iteration theorem for monotone systems [75, 195] applied to (2). The proof is in Appendix A. It rules out a class of defects that hand-written visitors can have: breadth-first and depth-first visitors that disagree, wiring that depends on hash order, and results that change between runs.

Proposition 7 (Incremental maintenance)**.** *Represent a revision as a Z-set (a signed multiset) of elements and incidences, and a patch as a Z-set Δ. If a traversal is expressed in DBSP's operators (linear operators, bilinear joins, distinct and fixpoint), then its incremental form θ^{Δ}, a stateful operator that keeps the integrals of its inputs, satisfies $\theta(G + \Delta) = \theta(G) + \theta^{\Delta}_{G}(\Delta)$. For linear operators the cost is proportional to $|\Delta|$ [46].*

Service	Selector	Class	Output
Dependency injection	Bind consumer→provider; containment (inherited environment)	lattice (c)	Bind patch; findings: missing, ambiguous, cross-place
Navigation, deep links	Transition on Routes; route containment	Boolean/min-plus (b), fold (a)	route table; deep-link plan; unreachable-route findings
Rendering and layout	containment of Surfaces and slots	fold (a)	render plan realised by Compose or React hosts
Lifecycle	runtime containment; Bind order	fold (a)	activation and deactivation plan (providers first)
Capability and placement	capability ports; Places	lattice (c), solver (d)	eligible places; placement with certificate
Effects, authority and flow	Session, Channel, Access from entry points; Grant; labels	Boolean (b), lattice (c)	effect footprint per entry; ungranted effects; illegal flows
Code-push closure	required-together relations (Section 9.5)	lattice (c)	smallest compatible release region
Impact and tests	reverse reachability over connections and Access	Boolean (b)	affected scopes, invalidated evidence, tests to run
Telemetry	Session, Channel in a scope	fold (a), returning a patch	patch adding interceptor and observer roles
Data invalidation	References, Access, Derives	Boolean (b)	caches and views to invalidate
Lints	any structural predicate	(a)–(c)	findings with witnesses and coverage
Data graphs	References over graph-family datasets	(b), (c), compiled to the store	paths, cycles, cascades and visibility, with coverage

Table 8: Platform services as traversals.

Reaktor's design documents require that an incremental update equal a fresh analysis of the same input [293]. With Proposition 7 this is a theorem about the compiled traversal rather than a test of each implementation. Findings, injection plans, route tables and closures become views maintained under patches, as in differential dataflow [236] and build systems [251].

Proposition 8 (Coordination-freedom)**.** *In the model of relational transducer networks, a traversal over a graph partitioned across machines can be evaluated without coordination if and only if it is monotone in the graph [16, 167]. Non-monotone traversals ("no provider exists", "this route is unreachable", "no grant permits this effect") are evaluated as stratified programs over a consistent cut [56].*

The kernel labels each traversal monotone or stratified. It refuses to report a non-monotone result over an inconsistent cut, reporting *unknown* instead. This is the formal form of the rule that a missing observation is not evidence of absence.

8.3 Instances

Table 8 lists the platform services that are traversals in RG. Each was a subsystem, a class or a manual procedure in the current runtime.

Navigation. Routes and their transitions form a statechart [163]: route containment gives hierarchy, a container whose tabs keep separate back stacks gives orthogonal regions with history, and a modal return is a Transition in return mode carrying a typed result (Figure 13). Resolving a deep link is a path problem: find the shortest path of transitions, under their guards, from the current configuration to the target's chain of ancestors. Back stacks are runtime state over this structure. Because they are validated walks of the transition graph, a deep link can no longer produce a stack that no sequence of user actions could produce, which the current runtime does not check.

Rendering. A Surface owns a region of the interface, not every painted shape: text, layout and drawing inside it stay ordinary Compose or React code. Containers are surfaces with named slots, and screens are surfaces mounted at routes. Rendering is a fold over the active configuration. It passes constraints and theme down and receives sizes and readiness back, and its result is a render plan of keyed mounts that a Compose or React host reconciles. A surface binds to its component's ports, observing Channel signals and sending to mailboxes or sessions, so state and operations stay in the graph while presentation stays in the renderer. Because the fold is generic over slots, an arbitrary design is a different arrangement of surfaces and slots, not a copy of a framework container. When a surface's logic runs in a pushed region (Section 9.5), the render plan crosses the trunk as data: a keyed tree of mounts, typed by the surface contracts the host offers. The host's widget catalogue is a set of capability contracts. A pushed region can therefore use only the surfaces its host provides, and admission checks this, as in Redwood [52].

Traversals over data. Rows never enter the graph, but much data is itself a graph: follow relations, dependencies, prerequisites, bills of materials. A traversal over a dataset of the graph family is compiled into the store's own query language and runs where the data lives: recursive SQL for Postgres, Cypher [139] for a graph store such as Memgraph [239], SQL

```
val blockedBy = traversal<Set<TaskId>>("tasks.blocked-by") {
    over(db["tasks"], along = Task::dependsOn)
    start { it.status == Open }
    lattice(SetUnion)
    compileTo(Postgres)
}
```

Listing 2: A traversal over a dataset (proposed API) and the query it compiles to.

```
WITH RECURSIVE reach(task_id, dep_id) AS (
    SELECT t.id, d.depends_on
    FROM tasks t JOIN task_deps d ON d.task_id = t.id
    WHERE t.status = 'open'
  UNION
    SELECT r.task_id, d.depends_on
    FROM reach r JOIN task_deps d ON d.task_id = r.dep_id
)
SELECT task_id, array_agg(dep_id) AS blocked_by, bool_or(dep_id = task_id) AS cyclic
FROM reach GROUP BY task_id;
```

Listing 3: The recursive query that Listing 2 compiles to, written out by hand; `UNION` deduplicates, so the recursion terminates on cycles and reports them.

with window and array functions for an analytical store such as ClickHouse, or Datalog. Its algebraic class is unchanged. Theorem 6 therefore still holds, and Proposition 7 holds wherever the store offers a change feed. The kernel plans the query and the store executes it, so path finding, cycle detection, deletion cascades and visibility are each written once, not once per client and store. Listings 2 and 3 show one traversal and its compiled form.

Dependency injection and other containers. Resolution is the lattice traversal of Table 8. It searches the scope chain for compatible providers, reports a missing or ambiguous provider as a finding with a witness, and never depends on construction order. Foreign containers such as Koin, Hilt or Spring are adapters: an opaque Scope whose beans are Bind providers, imported with provenance. Interoperating with a framework therefore no longer requires the kernel to depend on it.

8.4 Pure traversals, effectful executors

Traversals are pure. Effects belong to *executors*, which apply plans under explicit authority. This preserves the design rule that visitors are pure, bounded interpretations [293] and that rewrites return patches, and still lets everything that the current runtime hard-coded become graph-derived:

- **Structured concurrency.** The lifecycle executor applies activation plans. Each Activation gets a coroutine scope that is a child of its parent activation's scope, so the runtime containment forest *is* the job tree [105]. Cancellation propagates down the forest, and failure propagates up according to the scope's supervision facet [20]. The dispatcher is a capability port offered by the place: the main thread on Android, the event loop in a browser, the microtask queue in Hermes. This replaces the current runtime's scopes fixed inside classes (a `SupervisorJob` and a default dispatcher per graph, with `GlobalScope` escapes) by scopes derived from the graph.
- **Reconciliation.** A reconciler is a traversal that computes $\mathit{desired} \ominus \mathit{observed}$, keyed by stable identity, plus an executor that applies the difference idempotently. For a surface subtree this is React's keyed reconciliation; for places and resources it is a Kubernetes controller [47]. Both are level-triggered, and both read the same graph.
- **Instrumentation.** An installed observer is a plan in the observation layer, attached to a revision and removed with it, never an untracked side effect.

The four categories of the design documents (read-only analysis, rewrite proposal, installed instrumentation and runtime execution) are thus queries, patches, observation plans and executors.

9 Change: patches, transactions, diffs and code push

9.1 Patches

Definition 9 (Patch)**.** A *patch* is a typed double-pushout rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ with monic legs that preserve identity on K, together with nested application conditions [159] and a base revision. Applying p at a match $m : L \to G$ constructs D as the pushout complement of $K \to L \to G$ and H as the pushout of $D \leftarrow K \to R$ (Figure 14).

Identifiers fix the match: m is the inclusion of the elements that L names, so applying a patch requires no subgraph-isomorphism search and costs $O(|L| + |R|)$ index lookups. The dangling condition makes deleting a node without its ports and incidences a rejected rule rather than a dangling reference. The primitive operations are adding and removing

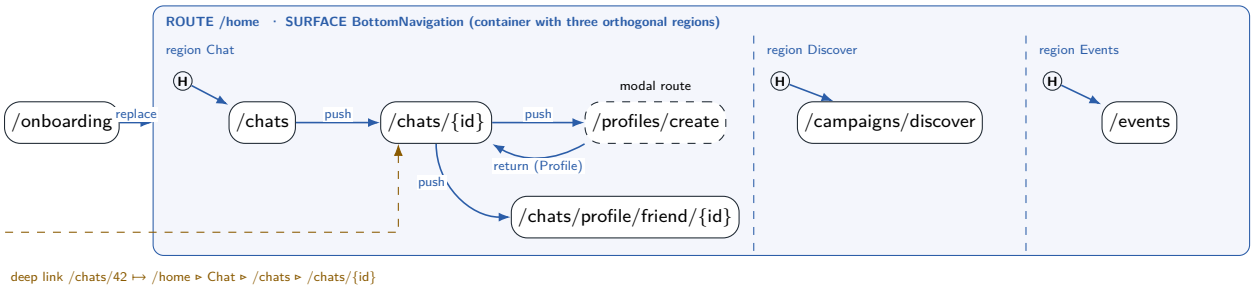


Figure 13: Navigation as a statechart over Routes and TRANSITIONS. Containment gives hierarchy; a container with three tabs gives orthogonal regions, each with a history state (H) that restores its back stack; a modal returns a typed result. A deep link is resolved as a shortest valid path, so every stack it produces is reachable by user actions.

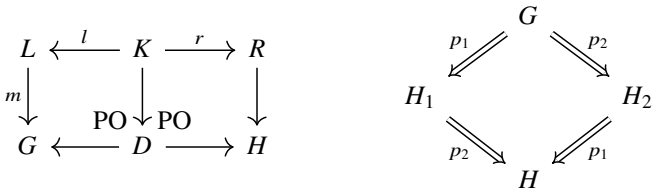


Figure 14: Left: applying a patch is a double pushout. Right: parallel-independent patches commute (Theorem 10); their merge is order-free and needs no review beyond each patch's own.

nodes, ports, hyperedges and incidences, re-parenting, and setting an attribute against an expected value. Every graph change is a composite of these, including the elaboration of source code, an agent's proposal, a migration and a release. Patches are data. They have identities and digests, and they live in the source layer as Artifacts.

Proposition 9 (Local preservation)**.** *For every primitive operation o there is a precondition pre_o, checkable in time proportional to the operation and its declared read set, such that* $\mathsf{WF}_{\mathsf{static}}(G) \wedge pre_o(G) \Rightarrow \mathsf{WF}_{\mathsf{static}}(o(G))$.

Validation is therefore incremental: a transaction checks each operation's precondition and never re-validates the whole graph. Global constraints (uniqueness, exclusive binding, acyclicity) declare predicate read sets, which commit re-checks. This replaces "apply, then check everything", which the stable-foundations audit rejects as a definition of validity [295].

9.2 Transactions

A *definition commit* is a transaction at the graph authority of one consistency domain (a workspace, or a partition of a large one). It carries a base revision, a patch and a read set of predicates. It commits if the base is current, or if no predicate in the read set changed since the base; this is optimistic concurrency with serializable outcome. Readers see either the old revision or the new one. The laws are those of the stable-foundations audit: a failed patch leaves state unchanged; a successful patch advances the revision and preserves $\mathsf{WF}_{\mathsf{static}}$; and at most one patch commits from a given base [295]. Four kinds of change stay distinct, because different authorities commit them:

definition commit (graph authority) · activation (host, under an epoch) · data transaction (a Store) · external effect (a provider).

A change that spans them is a plan executed as a protocol, with an outbox or a saga [140, 166]. It is never a distributed ACID transaction. The word "transactional" in R17 therefore means: transactional for definitions, and planned with receipts for everything else.

Refs and proposals. The graph authority keeps named *refs*, each pointing to a revision, and *proposals*, each a patch with a base revision, an author principal, its checks and a status. A branch, a worktree or an agent's line of work is a ref. Review and merge act on proposals, and Theorem 10 decides which of them merge without review. A data transaction at a store enforces the invariants that its datasets declare (Section 6) and rejects a violation with a witness. Figure 15 shows the path every change takes.

9.3 Concurrent change and merge

Theorem 10 (Commutation)**.** *Let p_1 and p_2 be patches applied to G at matches m_1 and m_2 with $m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2))$, so that neither deletes what the other uses, and suppose that each preserves the other's application conditions. Then p_2 applies after p_1 and p_1 after p_2, both orders give the same graph, and the parallel rule $p_1 + p_2$ applies to G directly.*

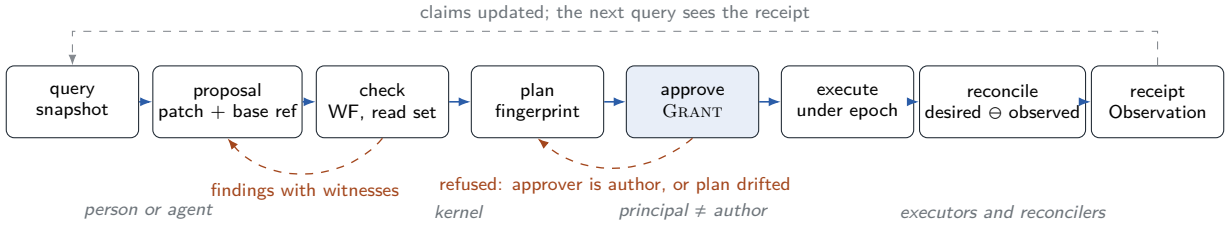


Figure 15: One path for every change, from a person or an agent alike. A rejected check returns findings; an approval binds the plan's fingerprint and can never come from the proposal's author. Execution happens under an epoch, and the receipt becomes an observation that the next query sees.

```
val addAuth = patch(base = ref("main"), author = principal("agent:reviewer")) {
    addIncidence(edge = "chat.api", role = Interceptor, port = "auth.verify", ordinal = 0)
    setAttribute("chat.timeline", Retention, expect = 7.days, to = 30.days)
}
```

Listing 4: A patch as data (proposed API).

This is the Local Church–Rosser and Parallelism theorem for adhesive categories [102, 210], which applies by Proposition 1. With identity-preserving rules, "the same graph" means equal, not merely isomorphic. Three consequences follow. Merging independent changes from two people or agents is automatic and order-free. Conflicts are exactly the overlapping delete–use pairs plus violated application conditions, and for a finite rule catalogue they can be enumerated in advance as critical pairs. Application conditions over global predicates can still make two independent patches jointly invalid, for example two providers bound to one exclusive consumer. The commit's read-set check catches this case: individually valid is not jointly valid. Presentation state (layout, folds, comments) has no invariants and is merged by CRDTs instead, including the tree-move CRDT for regrouping [203, 310].

9.4 Graph diffs replace textual diffs

The diff between revisions G_1 and G_2 is the span $G_1 \leftarrow G_1 \cap G_2 \rightarrow G_2$, where the intersection is computed by joining on identifiers: an $O(n)$ merge of sorted identifier columns, or $O(|\Delta|)$ from the change log. A diff is itself a patch. Diffs therefore compose. They invert, because $R \leftarrow K \rightarrow L$ is again a rule. And they apply to any graph that contains their left side, which is the embedding theorem: a change to a region applies wherever the region is mounted. The categorical theory of patches [248] reaches the same structure for files. A reviewer of an RG diff sees typed changes, grouped by layer and ranked by risk. Examples are "SESSION `send` gains interceptor `Auth` at position 0", "Dataset `messages`: field `roomId` now references `rooms.id`", and "Scope `chat`: effect ceiling widened to `payments.*`". Textual diffs remain only for the opaque bodies of code entries, attached to the Components they implement. Listings 4 and 5 show a patch written by an agent and the diff a reviewer sees.

Each scope has one edit owner. For a *source-owned* scope, elaboration is a function from a source revision to a graph patch. The graph diff of a commit is then $elab(after) - elab(before)$, computed in CI and reviewed instead of the text. For a *graph-owned* scope (Blueprint behaviour, deployment, data schema, policy), the patch is primary and source is generated. Edits from both sides must satisfy the lens laws on the supported subset [138]. Moving ownership between the two is an explicit migration patch. This is how graph diffs replace git diffs: version control stores and transports bodies, while review, merge, impact analysis and evidence invalidation work on patches. As agents write more of the code and people read less of it, the patch becomes the artefact that people review.

9.5 Code push as a region release

A *region* is a scope selected for replacement. Its *compatibility closure* is the least fixpoint

$$C^* = \mu X.\, S \cup Req(X), \tag{3}$$

where $Req(X)$ adds four kinds of element that must change with X: elements joined to X by a BIND or a move or borrow incidence without an adapter, since in-process references do not cross a version boundary; participants of sessions whose

```
~ Session  chat.api        + interceptor auth.verify @0       risk medium
~ Channel  chat.timeline   retention 7d -> 30d                risk low
  impact   2 scopes; claim timeline.retention-budget is stale
  merge    commutes with 3 open proposals; conflicts with none
```

Listing 5: The graph diff of Listing 4, as a reviewer sees it.

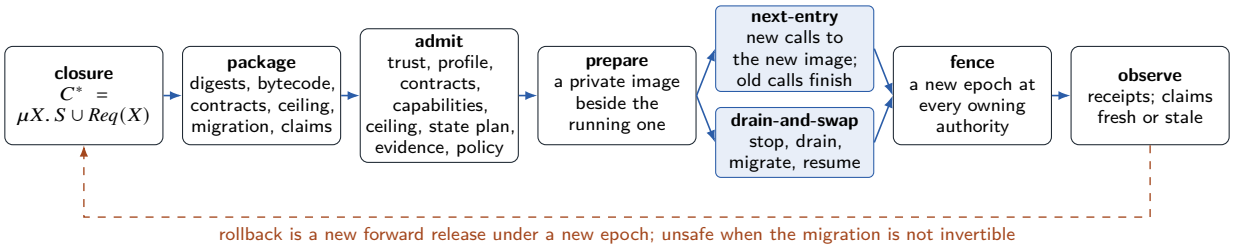


Figure 16: Code push as a region release (Section 9.5). The closure is a least fixpoint computed by a traversal. Admission is decided on each host, and activation switches one local root, in one of two modes. Writes from a retired generation are refused at the owning authority by epoch.

global type changes; datasets whose schema changes, with their accessors, unless a migration preserves compatibility; and regions fused with X into one compiled artefact. *Req* is monotone, so C^* is computed by a traversal (Theorem 6). This generalises the closure bound of Reaktor's engine preprint [288] to hyperedges: a multi-party session cannot be split by a cut without its relation identity. In Figure 2 the region `social-ui` is closed in this sense. Its binds leave it only through FFI adapters whose contracts the installed app binary fixes; it calls no session and owns no dataset; and it compiles to its own bytecode. It therefore ships without a store release.

A *release* is an Artifact containing:

- base and candidate digests, and the definition patch;
- the code artefacts: Kotlin/JS compiled to Hermes bytecode for a pinned engine version;
- the exported contract fingerprints, and the capability manifest of host imports;
- the effect ceiling, the state migration with its pre- and postconditions, the activation mode and the rollback class;
- claims with evidence, and a signature with update-framework metadata [305].

A host *admits* a release when all of the following hold: the bytes are trusted; the host profile (Section 13.1) is compatible; the boundary contracts are compatible; the required capabilities are available; the effect ceiling lies within a grant; the state plan is valid; the evidence is fresh for the candidate digest; and the distribution policy permits the release. The last condition is decided by platform rules, not by the graph [19]. The host prepares the image privately and publishes it by switching one local root. In *next-entry* mode, new invocations use the new image and old ones finish on the old. In *drain-and-swap* mode, the host stops admission, drains, migrates and resumes. Callbacks and handles of a retired generation are rejected or routed to the pinned generation. Writes are fenced by epoch at the owning authority. Rollback is a new forward release under a new epoch; a release with a non-invertible migration declares that rollback is unsafe. Dynamic software updating [170], version consistency [262] and Erlang release handling [129] are the precedents. Corollary 5 states the semantic condition on the boundary, the bounded models below check the two protocols this activation relies on, and Listing 6 specifies the activation itself. Figure 16 summarises the sequence.

Release in TLA$^+$. Listing 6 specifies next-entry activation. A call binds to the generation that is active when it enters, and it may leave a deferred callback that outlives it. A release activates the next generation. Retiring the oldest generation fences its epoch at the owning authority, and with draining it waits for that generation's calls. Two invariants state what code push must never do: accept a write from a retired generation, or refuse the write of a call that was admitted. TLC checks three generations and three calls. With fencing and draining, both invariants hold over 3,858 states. Without the fence, a callback of a retired generation writes after six steps; without draining, an admitted call's write is refused after four (Table 34).

10 RGL: the Reaktor Graph Language

RGL is RG embedded in Kotlin common code. It has four sublanguages that share one syntax and one identity fabric. *Definitions* elaborate scopes into patches. *Behaviour* is code entries that perform distributed operations. *Queries* are traversals. *Changes* are patches, rewrites, migrations and releases. The language is defined by RG's abstract syntax and semantics, not by Kotlin (Section 10.8).

10.1 Phases

Every RGL program passes through phases whose effects and authorities differ:

$$\text{elaborate} \to \text{check} \to \text{plan} \to \text{compile} \to \text{admit} \to \text{activate} \to \text{observe}.$$

Elaboration runs ordinary Kotlin (loops, functions, generics) over pinned inputs and produces a patch. It has no production effects: files, clocks, randomness and network access are excluded or declared as build inputs, so elaboration is deterministic and its patch has a digest. *Check* is $\mathsf{WF}_{\mathsf{static}}$. *Plan* chooses placements and bindings and computes closures. *Compile* produces JVM, native and JavaScript artefacts, and Hermes bytecode for pushable regions. *Admit* evaluates

```
---------------------------- MODULE RegionRelease ----------------------------
EXTENDS Naturals

CONSTANTS MaxGen, MaxCalls, Fence, Drain

VARIABLES active, retired, fence, calls, deferred, writes, entered

vars == <<active, retired, fence, calls, deferred, writes, entered>>

Init ==
    /\ active = 1
    /\ retired = {}
    /\ fence = 0
    /\ calls = {}
    /\ deferred = {}
    /\ writes = {}
    /\ entered = 0

Write(g, kind) ==
    [gen |-> g, kind |-> kind, accepted |-> g >= fence, retiredGen |-> g \in retired]

Enter ==
    /\ entered < MaxCalls
    /\ calls' = calls \cup {[id |-> entered, gen |-> active]}
    /\ entered' = entered + 1
    /\ UNCHANGED <<active, retired, fence, deferred, writes>>

Defer(c) ==
    /\ c \notin deferred
    /\ deferred' = deferred \cup {c}
    /\ UNCHANGED <<active, retired, fence, calls, writes, entered>>

Finish(c) ==
    /\ calls' = calls \ {c}
    /\ writes' = writes \cup {Write(c.gen, "call")}
    /\ UNCHANGED <<active, retired, fence, deferred, entered>>

Fire(d) ==
    /\ d \notin calls
    /\ deferred' = deferred \ {d}
    /\ writes' = writes \cup {Write(d.gen, "callback")}
    /\ UNCHANGED <<active, retired, fence, calls, entered>>

Release ==
    /\ active < MaxGen
    /\ active' = active + 1
    /\ UNCHANGED <<retired, fence, calls, deferred, writes, entered>>

Retire(g) ==
    /\ g < active
    /\ g \notin retired
    /\ \A h \in 1..(g - 1) : h \in retired
    /\ Drain => \A c \in calls : c.gen # g
    /\ retired' = retired \cup {g}
    /\ fence' = IF Fence THEN g + 1 ELSE fence
    /\ UNCHANGED <<active, calls, deferred, writes, entered>>

Next ==
    \/ Enter
    \/ \E c \in calls : Defer(c) \/ Finish(c)
    \/ \E d \in deferred : Fire(d)
    \/ Release
    \/ \E g \in 1..MaxGen : Retire(g)

Spec == Init /\ [][Next]_vars

NoWriteFromRetired == \A w \in writes : w.accepted => ~w.retiredGen

NoLostCallWrite == \A w \in writes : w.kind = "call" => w.accepted
=============================================================================
```

Listing 6: Region release with next-entry activation, in TLA$^+$ (`anc/RegionRelease.tla`).

Model	Bound	States	Trans.	Result
Handover, no fencing	2 owners, 1 write each, 1 handover	12	13	violated: send(A,1); handover(A→B,2); accept(A,1)
Handover, fenced at authority	same	15	19	holds
Handover, sender checks its lease	same	17	20	violated (same 3 steps)
Handover, no fencing	2 owners, 2 writes, 2 handovers	32	39	violated (same 3 steps)
Handover, fenced at authority	2 owners, 2 writes, 2 handovers	379	812	holds
Handover, sender checks its lease	2 owners, 2 writes, 2 handovers	40	51	violated (same 3 steps)
Handover, fenced at authority	3 owners, 2 writes, 3 handovers	4 555	13 928	holds
Effect, no deduplication	1 effect, ≤3 attempts, loss, 1 duplicate	23	27	violated: applied twice in 4 steps
Effect, deduplicated by effect id, timeout read as UNKNOWN	same	87	268	holds
Effect, timeout read as NOT_APPLIED	same, deduplicated	83	248	violated: client concludes “not applied” after commit

Table 9: Exhaustive breadth-first exploration of two activation protocols (Appendix C). The invariants are: (i) a write is accepted only under the current term of its current owner; (ii) an effect is applied at most once, and a client never concludes that an applied effect was not applied. The script is included with the paper. For a model that holds, the counts cover the whole bounded state space; a search that finds a violation stops at the first one, so its counts are partial. Table 34 gives the TLC results for the same protocols.

```
val Chat = scope("chat", owner = Source, updateUnit = true) {
    val api      = require(Session.call<MessagingApi>())
    val timeline = channel<RoomEvent>("timeline", Signal)
    val inbox    = channel<ChatCommand>("inbox", Mailbox)
    val db       = store("chat-db", Transactional) {
        dataset<Message>("messages") { key(Message::id); references(Message::roomId, "rooms") }
    }
    component("actor", ::ChatActor, profile = Exclusive) {
        consumes(inbox); produces(timeline); calls(api); writes(db["messages"])
    }
    val list   = route("/chats")
    val detail = route("/chats/{id}", ChatArgs.serializer())
    surface("list", ::ChatListScreen, at = list)   { observes(timeline); navigates(list to detail, Push) }
    surface("chat", ::ChatScreen,     at = detail) { observes(timeline); produces(inbox) }
    export(detail)
}
```

Listing 7: A self-contained feature scope (proposed API).

$\mathsf{WF}_{\mathsf{act}}$ and the release predicate on each host. *Activate* publishes. *Observe* joins runtime facts back to the graph by identity. A loop in behaviour updates local state; it never publishes a revision. A revision is published only by a committed patch.

10.2 Definitions

Listing 7 declares the chat feature of Figure 1, with one difference: the message store belongs to the scope, so the actor writes it directly rather than through the messaging service. The scope has a boundary session client, a signal channel (fan-out, latest value), a mailbox channel (competing consumers, one consumer), a store, an exclusive component, two routes and two surfaces. Every call creates typed elements; every declared name becomes a stable identifier within its scope; every port is typed by family, role and contract. The scope is self-contained: its only external need is a boundary Session client for `MessagingApi`.

10.3 Behaviour

Behaviour is ordinary Kotlin in *code entries*: named, registered functions with a receiver that offers the distributed operations of Table 10. The unit transferred to another place is a code reference with input values and delegated capabilities. It is never a closure: a closure may capture a mutable object, a coroutine or an authority that cannot move. Listing 8 shows an effect committed together with its outbox record in one transaction at one authority; the record is published only after the commit. It also shows a fan-out to shard owners with an explicit join policy, followed by a call to a ranker that a Python provider implements.

10.4 Queries and changes

Traversals are declared with their selector and their algebra, so the kernel can check monotonicity and compile them incrementally (Section 8.2). Rewrites are traversals that return patches. Listing 9 computes reachable routes and installs

```
@Entry("chat.send")
suspend fun RglScope.send(cmd: SendMessage): Outcome<MessageId, SendError> =
    atomic(chatDb, idempotency = cmd.effectId) {
        val id = insert(Message(cmd.room, cmd.text))
        outbox(MessageSent(id, cmd.room))
        id
    }

@Entry("search.rank")
suspend fun RglScope.rank(q: Query): Ranked = distributedScope(failure = CollectAll) {
    val parts = q.shards.map { s -> spawn(SearchShard, ShardQuery(q.text, s), at = ownerOf(s)) }
    call(ranker) { rank(parts.awaitOutcomes().successes(onUnknown = Exclude).flatten()) }
}
```

Listing 8: Code entries with distributed operations (proposed API).

Operation	Family	Required meaning
`call(port) {…}`	Session	Returns an Outcome that separates a domain rejection, a failure and an unknown effect; carries a deadline and an idempotency key
`emit`, `receive`	Channel	Admission at the channel's stated point; acknowledgement and replay per the channel's attributes
`fire(transition)`	Transition	Enabled only if the guard and the join policy hold; durable transitions persist before acknowledging
`spawn`, `await`	runtime containment	A child activation under the current scope; the join policy is explicit; a timeout is not evidence of termination
`atomic(store)`	Access	A transaction at one named authority; the body may use only that authority's operations and its outbox
`submit(work)`	Session (durable)	Returns a handle only after the durable authority has accepted the request
`observe(ref)`	Observes	Facts with provenance, coverage and freshness

Table 10: Distributed operations available to code entries.

a tracing interceptor on every call in a scope. Its second half is how auth, telemetry, rate limiting and similar concerns are "bolted on" (R5). Each is a typed, reviewable patch, so its effect is visible in the graph diff.

10.5 Graph types as Kotlin types

RGL encodes as much of a port type as Kotlin's type system can carry. Many errors are then ordinary type errors, reported by the Kotlin compiler before any RG rule runs; the plugin checks the rest.

- *Ports.* A port handle is typed by family, role and contract, as in `Port<Session, Client, MessagingApi>`. A connection operation accepts only ports of its own family and roles that the family allows, so connecting a Session client to a Channel, or giving a call session two servers, does not compile (rule Inc of Figure 9, and W3).
- *Contracts.* A contract is a Kotlin type, usually a sealed interface of serializable classes. Its fingerprint is the digest of the canonical form of its serial descriptor [184]. The canonical form includes every serial name that reaches the wire, so renaming a class changes the fingerprint unless `@SerialName` pins the old name, and any change of schema changes it.
- *Outcomes.* A Session call returns a sealed Outcome with four cases: success, domain rejection, failure and UNKNOWN. A `when` over it without an `else` branch must be exhaustive, so code that forgets UNKNOWN does not compile, and the plugin flags `else` branches and helpers that discard it silently.
- *Effect ceilings.* The receiver of a code entry is indexed by its ceiling, `RglScope<E>`; Listing 8 omits the parameter. An operation whose effect exceeds E is not a member of that receiver, so W10 is a type error at the call site.
- *Ownership.* A borrow port hands its body a scoped value, as the parameter of a lambda that the plugin's escape analysis forbids to escape (W9). Kotlin has no linear types [349], so the plugin checks that a move value is consumed exactly once only on statically known paths, and reports the rest.
- *Labels.* Data from a labelled source arrives wrapped, as `Labeled<T, L>`. Only a declassifier held by an authorised

```
val reachable = traversal<Boolean>("routes.reachable") {
    select(Transition, from = Pre, to = Post); start { it == root }
    lattice(BooleanOr)
}
val traceCalls = rewrite("telemetry.trace-calls") {
    forEach<Session>(where = { it.protocol == Call && tracer !in it.interceptors }) { call ->
        addIncidence(call, role = Interceptor, port = tracer.port, ordinal = 0)
    }
}
```

Listing 9: A query and a rewrite (proposed API).

principal unwraps it, and the plugin tracks labels through bodies on statically known paths (W11), in the spirit of decentralized labels [259].

What Kotlin cannot express (hyperedges with several parties, placement, layers and the global rules) is checked on the elaborated graph. Each finding is reported at the source span that created the element, so it appears in the IDE next to the code that caused it.

10.6 Compilation and the Hermes host

The Kotlin compiler keeps its job. Symbol processing (KSP) extracts declarations, registries and FlexBuffers codecs. A compiler plugin performs the body-level checks that symbol processing cannot: capture analysis for code entries, effect summaries, and lifting entries out of closures. The Kotlin/JS IR backend emits an ES2015 module, and `hermesc` compiles it to bytecode for a pinned engine version [180, 241]. On a device, the native host runs its own RG runtime. A pushed region runs in a Hermes engine with a second runtime instance, connected to the host through a JSI trunk whose typed host imports are exactly the region's capability ports [289]. The engine is a runtime place nested in the app's process. Its boundary ports are Exported through the trunk, and Corollary 4 is what makes mounting it there sound. Section 13.4 describes the reference implementation of these steps.

This requires a kernel without platform dependencies. RG removes each obstacle found in the current runtime:

- identifiers are minted through a capability rather than `crypto`;
- time comes from a clock capability, and scheduling from a dispatcher capability that maps to the Hermes microtask queue;
- the kernel depends on no injection framework, HTTP client or database;
- ports match by contract fingerprint rather than by `KClass`, so separately compiled bundles connect.

The host profile pins the compiler, the engine version, the module format, numeric semantics, the coroutine dispatcher, interrupt and budget mechanisms, and teardown. It is qualified by tests on each target. Cash App's Zipline and Redwood run Kotlin/JS in an embedded QuickJS behind typed service interfaces and drive Compose-based UI from it [52, 53]. They are the closest industrial precedent. RGL generalises the unit from an interface to a typed subgraph, with a closure computation, an admission predicate and fenced activation.

10.7 Compiling with the whole graph

A compiler that sees one module must be conservative at every interface. It cannot know which implementation an interface call reaches, which fields a remote reader uses, whether a buffer escapes, which code is dead, or how often a path runs. Each of these is a fact about the system, and the graph records it. RGL therefore connects the Kotlin compiler and the graph in both directions (Figure 17):

- *Source to graph.* A build step runs the compiled definitions, which elaborates them into a patch: elements, ports, contracts, effect summaries and Access footprints. The K2 frontend [182] analyses bodies but does not execute them, so its checkers report the rules that a single module can decide, and the rest run on the elaborated graph.
- *Graph to compiler.* After planning, a *compilation plan* records the facts that the plan has fixed: resolved bindings, placements, the consumers of every port, release boundaries and profiles. The plugin's backend extension specialises code to it.

The compilation plan is a FlexBuffers file pinned by digest, and it is an input of the build like any source file. Builds therefore stay deterministic, and an artefact's digest identifies both its code and the facts it was specialised to. The separation is Halide's: the definition is the algorithm and the plan is the schedule [285], so one definition compiles differently for a phone, an edge isolate and a cluster. Graph-owned scopes, edited in the workbench rather than as text, take the same path: Kotlin is generated from the graph and compiled with the rest. Unlike Blueprints, which run on a virtual machine (Section 4), they then share the compiler, the optimisations and the debugger of handwritten code.

Optimisation domains. A fact may be used only where it cannot change behind the compiler's back. A release region is not such a place on its own: admission is decided per host (Section 9.5), next-entry activation runs two generations side by side, and versions coexist during a rollout (P13 in Section 16). The domain of an optimisation is therefore the set of elements of one release region that are admitted together on one host, at one generation. Admission checks their digest and the fingerprints at their boundary before activation, so every structural fact among them holds for as long as the code runs: which provider a Bind resolves to, which of its consumers share the host, which place each component occupies. Inside that set the compiler may assume a closed world, as whole-program compilers do [88, 352]. Any connection that crosses hosts, generations or an installed app binary keeps only its contract, because the other side can change independently, even when it belongs to the same region. Region boundaries are barriers in the same way. Optimising across one requires fusing the two regions, and the closure of Equation (3) then releases them together. A small region can be pushed quickly but admits fewer optimisations; a large one is optimised more but is released as a unit. The planner reports this trade for each candidate region.

The optimisations. Table 11 lists thirteen, each with the graph fact it needs. Four illustrate the pattern.

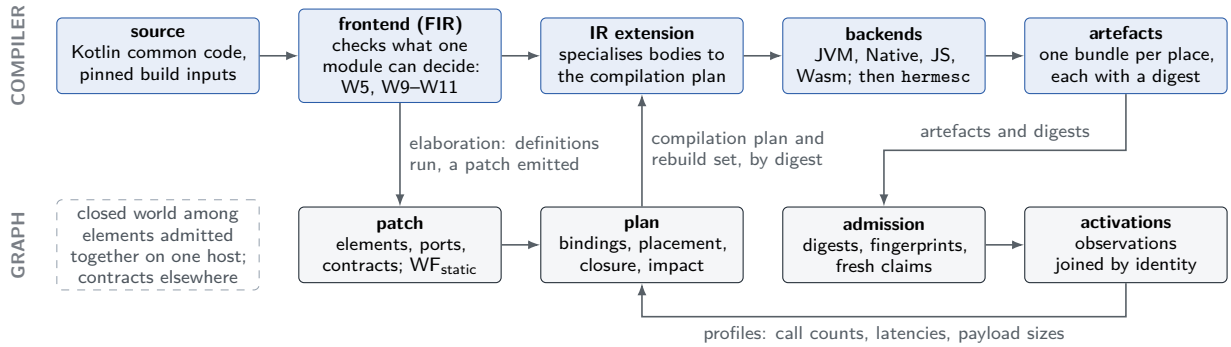


Figure 17: The Kotlin compiler and the graph in one loop. A build step runs the definitions, which elaborates them into a patch, and the frontend checks what one module can decide. The planner fixes bindings, placement and the release closure, and hands the compiler a compilation plan pinned by digest. The backend specialises code to it, admission checks the artefacts, and observations return as profiles for the next plan. Among the elements of a region admitted together on one host, the compiler may assume a closed world; across hosts, generations and regions, only contracts.

- *Static injection.* The injection traversal resolves every BIND at plan time. The compiler emits constructor calls in dependency order, as Dagger does within one program [150], but across the whole region and with the graph's scope chain. Nothing is looked up at run time, and a missing provider is a compile error with a witness.
- *Specialised sessions.* A SESSION is realized according to its placement. If the plan puts both ends in one process and one generation, and the request is deeply immutable or moved, the stub passes the request object without encoding it, so no alias can arise. It keeps the deadline and the Outcome, and an exclusive server still receives the request through its mailbox, because the family's semantics does not depend on the realization. Service Weaver makes the same choice between local calls and remote procedure calls at deployment time [143]. If the ends are in different places, the stub uses the codec generated for exactly that contract fingerprint.
- *Fusion.* Consider a chain of pure or stream components in one place and one supervision scope, linked by CHANNELS that each have one producer, one consumer, no durable buffer, no observer, lossless back-pressure and no time windows or watermarks. Removing such a buffer changes neither what is delivered nor when a failure is contained. The compiler fuses the chain into one loop, as stream fusion removes intermediate lists [77] and FlumeJava fuses parallel stages [55]. Attaching an observer to an intermediate channel is a patch. The patch makes the fused artefact stale, and the region is rebuilt unfused.
- *Codec pruning.* The plugin records, for each body, which fields of each contract it reads; forwarding a value, logging it or comparing it counts as reading every field. When the producer and every consumer of a transient channel run on one host at one generation, these summaries show which optional fields no reader uses, and the encoder skips them. A durable or replayed channel keeps every field, because a consumer added later may read them, and so does any channel with a reader on another host.

Specialising the runtime. A generic RG runtime interprets the graph: it looks up providers, dispatches calls through ports and encodes payloads by contract. Specialising an interpreter to a fixed program yields a compiled program. This is the first Futamura projection [188], and Truffle and Graal use it to compile guest-language programs by partially evaluating their interpreters [355]. A compilation plan is that specialisation, applied to the elements of a region on one host. Generated construction code completes it. The injection graph, registries and codecs are built by code generated at compile time, the counterpart, for Workers, ART and Hermes, of the heap image that GraalVM Native Image builds for a closed world [352]. A Worker, an Android process or a Hermes engine then starts without scanning, reflection or wiring.

Profiles from observations. Observations are joined to elements by identity (Section 5.4). Call counts, latencies and payload sizes per port are therefore graph data with provenance and freshness. Profile-guided optimisation [283] then works at the scale of the system:

- hot local calls are inlined;
- cold code is split into chunks that the web and Hermes hosts load lazily;
- codecs get fast paths for the commonest shapes;
- the placement solver learns which sessions are chatty enough to co-locate.

A profile is evidence, so it goes stale like a claim. Inlining and layout chosen from a stale profile change performance, not meaning. Code splitting and co-location are different: a lazily loaded chunk must be available when its host is offline, and co-location changes failure domains, so both are changes to the plan that admission checks, not choices the compiler makes alone.

Optimisation	Graph fact it relies on	Effect	Precedent
Static injection	Bind resolution by the injection traversal (Table 8)	constructor calls in dependency order; no container at run time; a missing provider is a compile error	Dagger [150]
Devirtualisation, inlining	a local Bind whose provider is unique in the region	direct calls that the backend can inline	class hierarchy analysis [88]
Dead-code elimination	ports without incidences; components unreachable from entries, routes and schedules	smaller bundles per Worker, app and engine; host capability ports that pushable regions may need are kept	R8 [152]
Session specialisation	the realization and placement of each Session	in-process stubs without codecs; one stub per kind of binding	Service Weaver [143]
Codec specialisation	contract fingerprints; per-body summaries of the fields each reader uses	generated FlexBuffers readers and writers; unread optional fields pruned among readers on one host	kotlinx. serialization [184]
Fusion	one-producer, one-consumer Channels between pure or stream components of one place and supervision scope, with no durable buffer, observer or time window, and lossless back-pressure	one loop, without intermediate buffers or tasks	stream fusion [77], FlumeJava [55]
Ownership	move, borrow and share on ports	no copies of borrowed buffers; moved values reused in place; W9 checked in bodies	escape analysis, linear types [61, 349]
Batching	Session attributes (batchable, idempotent read) and the call sites in one place	many calls to one place become one round trip	Haxl [235]
Partitioning by place	Places	one bundle per place; communication only across cut edges	TensorFlow [3]
Profile-guided optimisation	Observations joined by identity: call counts, latencies, payload sizes	inlining, code splitting, codec fast paths, co-location hints	Pettis and Hansen [283]
Incremental compilation	the impact traversal over a graph diff	only artefacts whose inputs or assumptions changed are rebuilt	ThinLTO [186], Bazel [331]
Closed-world initialisation	the elements admitted together on one host: entries, registries, codecs, injection plan	construction code generated at compile time; no scanning or reflection at start-up	Native Image [352]
Runtime specialisation	the whole compilation plan	the generic runtime specialised to one region	partial evaluation [188], Truffle [355]

Table 11: Optimisations for which the graph supplies the facts. Each is sound among the elements of a region admitted together on one host and generation, under the side conditions stated in the text; a connection that crosses hosts, generations or app binaries keeps only its contract.

Incremental compilation from graph diffs. The impact traversal (Table 8) computes which artefacts a patch affects:

- a body change that leaves the contract fingerprint unchanged rebuilds its module and every artefact that inlined that body or pruned a codec on its behalf;
- a contract change rebuilds the module and its dependents;
- a placement change rebuilds stubs and bundles, not bodies.

This is the summary-based scheme of ThinLTO [186] and the action graph of build systems [251, 331], with the system graph as the summary index. Each optimisation also records its assumptions as dependencies on elements. A patch therefore invalidates at least the artefacts whose assumptions it breaks. This is the mechanism that marks claims stale (Section 15). It is also the static counterpart of dynamic deoptimisation, which replaces optimised code when code that it inlined is edited [172]. Listing 10 shows a compilation plan as a reviewer sees it.

What authors declare. The compiler can use only facts that the graph states. Purity is a component profile; batching and idempotence are attributes of a Session; and optional fields are part of a contract. These already exist for checking, and the compiler reuses them. One attribute is new: a region is either sealed, or open to mounts at run time. An open region keeps more boundaries and receives fewer optimisations.

```
compilation plan  region chat  release chat@17  place edge  revision r-9f2c
  bind      actor.clock  -> isolate clock       resolved; call inlined
  session   actor.api    -> MessagingApi#7f3a    region boundary: stub and codec, no inlining
  channel   inbox        -> actor                mailbox keyed by room; codec ChatCommand#5d10
  codec     RoomEvent#91c2                       writer generated; nothing pruned: readers on phones
  init      registries, codecs, injection plan   built at compile time
  assumes   14 elements; a patch to any of them makes this plan stale
```

Listing 10: A compilation plan for the chat region on the edge place (proposed format, rendered as text).

Table	Hot columns (bytes)	Cold columns (bytes)	Hot bytes per row
Node	kind (2), parent (4), pre-order index (4), subtree size (4), facet mask (8)	identifier (16), name (4)	22
Port	owner (4), family (1), role (1), ownership (1), QoS (4)	identifier (16), contract (4), name (4)	11
Hyperedge	family (1), scope (4), attribute offset (4)	identifier (16)	9
Incidence	edge (4), end (4), role (1), position (2)	–	11
Indexes	hyperedge offsets (4 per hyperedge); port to incidence (4 per port); ports and children per owner (4 per node)	element to incidences, for assertions (4 per incidence)	–

Table 12: Kernel tables and column widths. Hot columns are what traversals read; cold columns hold identity, names and attributes.

Soundness and cost. Optimisation must not change meaning. The generic runtime, which interprets the graph, is the reference semantics, and a compiled region must refine it in a sense that also preserves progress: every behaviour of the compiled region, including what it refuses and whether it diverges, must be one that the interpreted region allows. Trace refinement alone (Corollary 5) would accept a compiled region that deadlocks. The conformance suite runs its test vectors on both and so tests their equivalence on those vectors; it does not prove it. Admission accepts an optimised artefact only with that evidence. Optimised code must also stay explainable. Every transformation is recorded in the compilation plan with the elements it touched, and source spans survive it (Section 13.4). A frame in fused or inlined code therefore still maps to the elements and lines it came from. Whole-region optimisation costs build time, so it is enabled per release profile; debug builds keep the interpreted path. Kotlin's compiler plugin interface is not stable and changes with compiler releases [181], so the plugin is pinned in the host profile and requalified with each compiler upgrade.

10.8 Portability

RGL Core consists of the abstract syntax (Σ_{std} as data, the patch operations and the traversal intermediate form), the semantics of Section 7, and the wire protocol. The wire protocol, `rgp/1`, carries graph messages in the frames of Reaktor's FFI layer, encoded with FlexBuffers [148, 289]; Section 12 defines it. A conformance suite of graph test vectors covers judgments, patch results, traversal results and specification traces. A host language can implement RGL if it offers six things:

1. sum and product types, or a schema system, for contracts and codecs;
2. first-class functions or a table of named entries;
3. asynchronous concurrency with cancellation;
4. a deterministic binary codec;
5. for pushable regions, an embeddable interpreter or a portable bytecode;
6. a C ABI or an equivalent FFI.

Swift, TypeScript, Rust, C++ and Python could implement RG from the same abstract syntax. Reaktor implements it once, in Kotlin, and uses Swift, C and TypeScript only as glue (Section 13.5). A system written in another language joins as a Component on its own Place, such as a Python model host behind a Session. Its internals stay opaque, and its boundary is typed like any other.

11 The kernel: a data-oriented realization

The C-set presentation of Section 5 is a database schema, and the kernel stores it as one. Each sort is a table, each structural arrow a column of foreign keys, and each facet a sparse set of typed columns (Figure 19). The design follows data-oriented practice in game engines [130, 157]: lay data out by access pattern, process it in batches, and keep identity separate from location.

11.1 Tables, rows and bytes

Table 12 gives the tables and the width of each column. Two choices dominate the layout.

- *Hot and cold columns.* A traversal reads only small integer columns: the 11-byte incidence row, the offset arrays and the family and role codes. Identifiers, names, contracts and attributes live in cold columns that a traversal never touches. Each cache line then carries useful data, and a scan of all incidences reads about 11 bytes per incidence.
- *Codes, not strings.* Kinds, families, roles and facets are small integer codes from the signature, and strings are interned once. Comparing a role is one byte compare, and a selector is a set of codes.

What fits where. The platform graph of Figure 2 has 226 elements, 198 ports, 100 connections with 205 incidences and 68 assertions. Its hot columns, indexes included, take about 15 KB, which fits in a first-level data cache. The 140 design graphs of the corpus take about 55 KB together. A system graph with a million nodes and ports and two million incidences needs about 50 MB of hot columns, and a scan over all its incidences reads about 22 MB, a few milliseconds

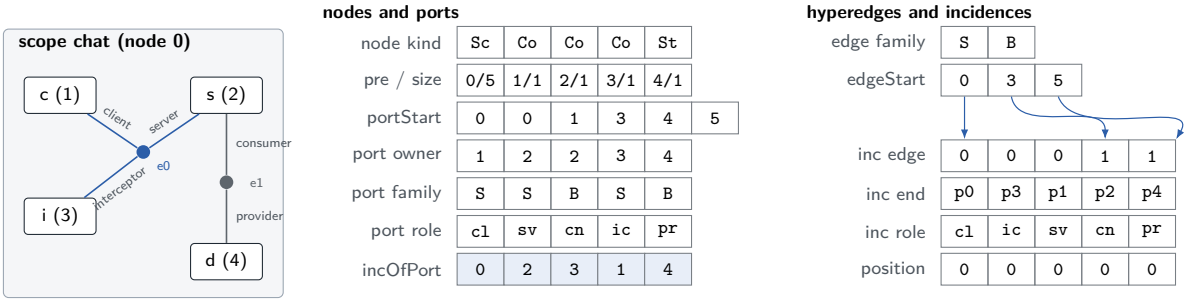


Figure 18: A call with an interceptor and one BIND, as the kernel stores them. Incidences are sorted by hyperedge, and `edgeStart` is the CSR index into them. Ports are contiguous per owner, found through `portStart`. Because every port is in at most one connection (W4), `incOfPort` (shaded) is a plain array. The pre-order index and subtree size make subtree tests two comparisons. Role codes: client, server, consumer, interceptor, provider; families: SESSION, BIND.

at memory bandwidth. Most traversals touch a selected family or scope, which is far less.

11.2 Layout and indexes

Layout. Inside a process, elements are addressed by dense slots with generation counters. A handle is (slot, generation), so a stale handle cannot alias a reused slot. Handles never leave the process: at boundaries, in persistence, in deep links and on the wire, elements are named by stable identifiers. Incidences are stored sorted by hyperedge, with a compressed-sparse-row (CSR) index from each hyperedge to its incidences and an inverse index from each element to its incidences. These are incidence matrices in the layout used by sparse linear algebra [194]. Ports are stored contiguously per owner, and children per parent. Facet attributes are sparse sets keyed by slot, which avoids one archetype table for every combination of overlays. Strings are interned and contracts live in a fingerprint table. On the JVM and Native the columns are primitive arrays. In Kotlin/JS, under V8 or Hermes, they are typed arrays, so a pushed region carries the same representation.

Containment as intervals. Each node stores its pre-order index and the size of its subtree, so "n lies in the subtree of s" is two integer comparisons. Rule W7 (boundaries) and scope selectors become interval tests, and a scope's elements are one contiguous range when nodes are stored in pre-order.

Nets as an array. Because a port is incident to at most one connection (W4), the index from ports to incidences is not a CSR structure but a single integer per port. Finding the connection behind a port is one load. Assertions may attach to any element many times, so their inverse index stays a CSR structure. Figure 18 shows the tables of a small graph.

Traversal loops. A selector compiles to sets of family and role codes and, for scopes, to pre-order intervals. Message passing is then a loop over CSR segments that reads two or three arrays in order. Listing 11 shows two loops: the server behind each call, and the worst-case latency of call chains in the max-plus semiring, evaluated in one pass that visits callees before callers. Neither loop allocates, and both read only hot columns. Lattice values are encoded as integers or doubles in parallel arrays, so a pass over a partition can run on its own thread and be merged by the lattice join (Theorem 6). Bitsets hold worklists and visited sets. On the JVM the inner loops are candidates for automatic vectorization.

11.3 Revisions, persistence and runtimes

Revisions. Columns are divided into immutable chunks. A revision is a root that points to chunk versions, so a commit copies only the chunks it changes [101]. Taking a snapshot is $O(1)$, and readers pin roots, which gives multiversion concurrency control [33]. Each commit also appends its delta to a change log as a Z-set of inserted and removed rows. The same delta feeds incremental traversals (Proposition 7), graph diffs and observers. Merkle digests are recomputed only along the containment paths that changed [240]. Named refs and proposals (Section 9) are kernel metadata over roots. Chunks hold 4,096 rows, so a row's chunk is its index shifted right by twelve bits, and a patch that touches a few elements copies a few chunks. A revision's root lists one chunk reference per column. Roots and chunks are reference-counted, and a chunk is freed when no pinned root reaches it. One writer per consistency domain commits; readers never lock, because what they read never changes.

Encoding and transport. A revision persists as a column dump: a header and each column's raw little-endian array, which can be memory-mapped and read without parsing. Messages that cross a boundary, such as patches, plans, findings, observations and FFI payloads, travel as FlexBuffers frames (Section 13.5). A receiver reads them in place from a JVM byte buffer, a JavaScript array buffer or C++ memory, so a pushed region, a Worker and a native host all decode the same bytes without copying.

Per runtime.

- *JVM and Android.* Columns are primitive arrays, and handles are value classes over a 64-bit integer, so nothing is boxed. Snapshots may live off the heap in memory segments.

```
class Columns(
    val portOwner: IntArray, val portFamily: ByteArray, val portRole: ByteArray,
    val portStart: IntArray, val edgeFamily: ByteArray, val edgeStart: IntArray,
    val incEnd: IntArray, val incRole: ByteArray, val incOfPort: IntArray, val incEdge: IntArray,
)

fun Columns.serverOf(edge: Int): Int {
    for (i in edgeStart[edge] until edgeStart[edge + 1]) {
        if (incRole[i] == SERVER) return portOwner[incEnd[i]]
    }
    return NONE
}

fun Columns.worstLatency(calleesFirst: IntArray, hop: DoubleArray, out: DoubleArray) {
    for (n in calleesFirst) {
        var worst = 0.0
        for (p in portStart[n] until portStart[n + 1]) {
            if (portFamily[p] != SESSION || portRole[p] != CLIENT) continue
            val e = incEdge[incOfPort[p]]
            worst = maxOf(worst, hop[e] + out[serverOf(e)])
        }
        out[n] = worst
    }
}
```

Listing 11: Two kernel loops over the columns of Figure 18 (proposed implementation).

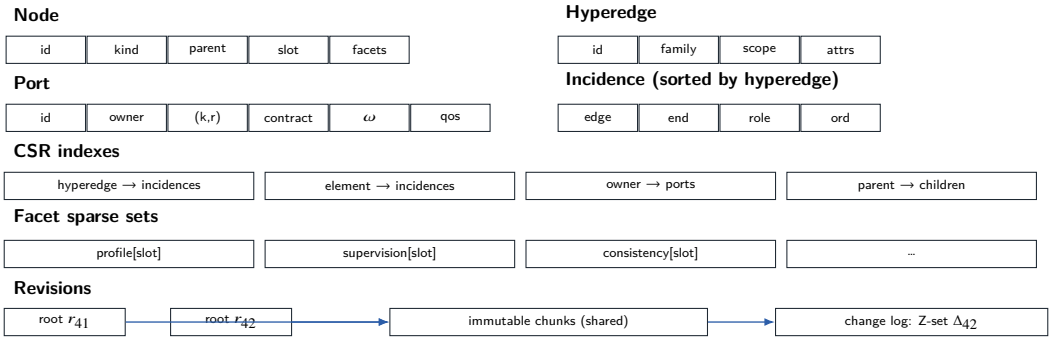


Figure 19: The kernel stores the C-set presentation directly: one table per sort, foreign keys as columns, CSR indexes for incidence and containment, facets as sparse sets, and revisions as persistent roots over shared immutable chunks with a Z-set change log.

- *JavaScript, V8 and Hermes.* Columns are typed arrays. Kotlin's 64-bit integers are emulated in JavaScript, so hot loops never use them: identifiers are four 32-bit lanes, and handles are a slot and a generation in two 32-bit arrays. Hermes runs precompiled bytecode, mostly without a just-in-time compiler, which is why hypothesis H3 sets a lower throughput target there.
- *Native and C++.* Columns are plain arrays that an FFI trunk can pin and hand to C++ as spans, so a native component reads the graph without a copy.

11.4 Plans and execution

Plans off the hot path. No call consults the graph. Activation compiles each consumer port into a direct reference (for BIND), a stub for its SESSION realization (local, FFI or RPC) or a channel endpoint. Interceptor chains are compiled into arrays. A local call through a BIND therefore costs one indirection and a check for interceptors. Observations name graph identifiers but never write to the graph. Because plans are compiled ahead of time, work already admitted continues through an outage of the control plane. Only operations that need a fresh grant, a reachable authority or a coordinated change stop.

12 RGP: the Reaktor Graph Protocol

The graph has many holders.

- A graph authority commits the definitions of one consistency domain (Section 9).
- Hosts on phones, edge isolates and servers admit releases and run activations under epochs (Section 9.5).
- Tools, agents and people query the graph and propose patches (Section 19), and verifiers attest claims (Section 15).
- Other authorities hold graphs of their own, for other partitions, environments, organisations or jurisdictions.

Operation	Current runtime (object graph)	RG kernel (columnar; design bound)
Resolve a consumer's provider	search up to the root with hashing; order-dependent	$O(1)$ through the compiled plan; resolution itself is a monotone traversal
Enumerate the connections of one family	full walk with allocation	$O(\lvert I_k \rvert)$ contiguous scan
Take a snapshot	full walk on every navigation change	$O(1)$: pin a root
Diff two revisions	not available	$O(\lvert\Delta\rvert)$ from the log; $O(n)$ by identifier merge
Canonical digest	not available	$O(n)$ once; $O(\lvert\Delta\rvert \cdot \mathit{depth})$ thereafter
Validate a change	not available (immediate mutation)	$O(\lvert \mathit{patch} \rvert + \lvert \mathit{read\ set} \rvert)$ (Proposition 9)
Mount a self-contained scope	imperative construction	pushout, $O(\lvert S \rvert + \lvert J \rvert)$
Find the connection behind a port	search the owner's relations	one load (W4)
Test subtree membership	walk up the parents	two comparisons (pre-order interval)
Serialise	reflection and JSON per object	column dump, $O(n)$

Table 13: Kernel operations. The middle column describes the current runtime's object graph as surveyed for Section 2. The RG column gives design bounds; hypotheses H3 and H4 (Section 20.4) measure the first three.

RGP, the Reaktor Graph Protocol, is how these parties exchange graph facts and changes. Its wire name is `rgp/1`, and it is part of RGL Core (Section 10.8).

Reaktor's specification of the protocol for the current runtime [290] fixed its shape, assembled from protocols that had solved parts of the problem (Table 15):

- one connection to many runtimes, with a session for each;
- typed commands and events, in domains that are enabled one at a time;
- list-then-watch cursors, in which a gap is an error;
- filters evaluated where events are produced;
- credit-based streams that never block the runtime they observe;
- a write path that prepares a change before it executes it.

That specification has not been implemented, and it describes the current runtime's graph. Its nodes, ports and binary edges are named by handles minted per session, its events are named after lifecycle callbacks, and its patches state whether they write a document, source code or a running system. This section re-founds the protocol on RG. Six changes follow from the model.

(i) *Names.* Elements are named by stable identifiers and revisions by digests, never by handles (Section 12.1).

(ii) *Reads are traversals.* A query ships a traversal to the data. A subscription is a traversal maintained incrementally, whose messages are Z-set deltas, and an expired cursor is reconciled by digests rather than by reading everything again (Sections 12.3 and 12.4).

(iii) *Each change goes to its own authority.* Proposals and commits go to the graph authority, and releases go to hosts. Rows and external effects never cross RGP; only their receipts do (Sections 12.5 and 12.6).

(iv) *Changes are sessions.* Every exchange that changes something is a Session with an effect identity bound to the request digest, a deadline, and an Outcome that includes UNKNOWN. Rule W5 applies to the protocol itself.

(v) *Every message is governed.* Grants, the effect lattice and labels decide what each principal may send and receive, including the graph's own metadata (A5, W11).

(vi) *Authorities federate through views, not routes.* Authorities exchange signed views filtered by labels. Nothing is merged, and no call is routed between them (Section 12.8).

RGP is itself an RG scope (Figure 20). Its endpoints are components on places, its conversations are Sessions, its streams are Channels and its authority is Grants. The rules of Section 6.5 and the templates of Table 34 therefore apply to the protocol that maintains the graph.

12.1 Parties, names and frames

Parties. Five kinds of party speak RGP:

- a *graph authority*, which commits the definition layer of one consistency domain and keeps its revisions, change log, refs and proposals (Section 11);
- a *host*, the runtime of a Place, which admits releases, runs activations and reports what runs;
- a *store*, which commits data and appears in RGP only through its receipts;
- a *client*: a workbench, a language server, CI, a verifier or an agent;
- a *peer authority*, which holds another consistency domain.

Every party acts as a Principal, is authenticated when a connection opens, and holds grants per domain.

Names. The kernel never lets a handle leave its process (Section 11). RGP therefore names elements, revisions, contracts, artefacts and activations as Definition 4 does: an element by its stable identifier, a revision by its digest $h(G)$, a contract by its fingerprint $\mathit{fp}(c)$, an artefact by its digest, and an activation by its definition, place, key and epoch. Five identities

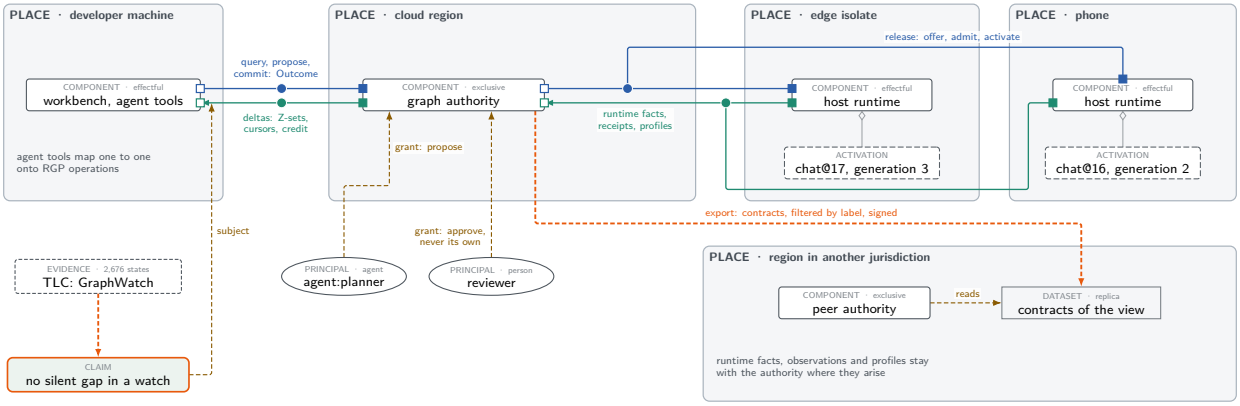


Figure 20: RGP in its own notation. Its endpoints are components on places, its conversations are SESSIONS and its streams are CHANNELS, each with its quality of service, and authority is a GRANT. The release is one multiparty SESSION with every host that runs the region. Orange marks what the protocol must never get wrong: a view exported to an authority in another jurisdiction, which only replicates it, and the claim that a watch never leaves a silent gap, attested by the TLC model of Listing 28.

stay apart: the definition, the contract fingerprint, the artefact digest, the configuration and the activation. Two builds with identical contracts and different bodies are therefore compatible and distinct. A build whose identity is unknown stays unknown, and its contract fingerprint never stands in for it. Lazy expansion keeps what handles were for. A read returns summaries (identifier, kind, name, digest and counts), and detail is fetched by identifier only for what a client shows. Identifiers survive a connection, so a client that reconnects refreshes what it holds instead of replacing it.

Frames. A frame is a FlexBuffers map with sorted keys (Listings 12 and 16). It names its message type by the fingerprint of that type's contract, and it names the conversation it belongs to. A frame that changes anything also carries an effect identity. The payload is decoded by position when both ends hold the same fingerprint, and by key when the sender's contract is compatible with the receiver's (Section 6). Otherwise the frame is refused; it is never decoded by position against a fingerprint it does not match. Bulk payloads (slices, deltas and batches of observations) are column chunks in the kernel's layout, so a receiver maps them without parsing. Patches, releases, claims and exported views are signed over their canonical bytes [32]; other frames rely on the integrity of the carrier. An unknown optional field passes through, and an unknown required field refuses the frame, as rule T4 says of facets.

12.2 Domains

Table 14 lists the domains. Each conversation is a connection of Σ_{std}, with a family, a quality of service and an effect class from the lattice of Section 6.4, and each domain is granted separately. Reading never implies watching, watching never implies reading payloads, and nothing implies changing. The six domains of the earlier specification survive in the table:

- its mesh domain is Connect;
- its graph and revision domains are Query and Watch;
- its wire domain and its domain for values become runtime facts and labelled samples in Observe;
- its edit domain is Change.

Release, Assure and Federate are new.

12.3 Reading: traversals at the data

A read is a traversal evaluated where the graph lives.

- `describe` returns a restriction view (Proposition 2) at a stated revision. The view holds the elements of a scope to a given depth and layer, and their ancestors, so that the forest condition holds. At the edge of the slice are *stubs*, which give only an identifier, a kind and a digest. A slice is therefore an open graph whose boundary is its stubs, and a client assembles a larger picture by gluing slices along shared identifiers, the composition of Section 7.
- `traverse` ships a traversal $\theta = (\sigma, V, \mu, \oplus, \upsilon, \beta)$ (Definition 8) to the authority or the host, which evaluates it there. Erlang's tracing filters events where they are produced, and the earlier specification adopted that idea for its selectors. RGP generalises it from a filter to any traversal, as a traversal over a dataset is compiled to the store that holds it (Section 8.3).
- The budget β replaces ad hoc limits. Every answer carries its coverage claim: the revision, the scopes visited, what the budget excluded, what labels withheld and which boundaries were opaque.
- A read across consistency domains is evaluated at a revision vector, one revision per domain, as in Definition 10. A

Domain	Messages	As a connection of Σ_{std}	Effect class	Rules
Connect	`hello`, `attach`, `detach`	SESSION, duplex	read	authentication; grants per domain; limits stated at `hello`
Query	`describe`, `resolve`, `traverse`	SESSION call	read	labels filter before encoding; coverage on every answer
Watch	`subscribe`, `delta`, `reset`, `reconcile`	SESSION stream with credits; deltas in order, replayed from a cursor	live read	never blocks the producer; every gap visible
Change	`propose`, `check`, `approve`, `commit`, `lookup`	SESSION call; effect identity keyed with the request digest; outcome includes UNKNOWN	non-production write	no self-approval (A5); read set checked at commit
Release	`offer`, `admit`, `activate`, `retire`, `lookup`	multiparty durable SESSION with the hosts	the join of the release's effects	each host decides admission; stores enforce fences
Observe	runtime facts, receipts, profiles, samples	fan-out CHANNEL, at least once, ordered per key, bounded retention	live read	a sample needs a grant for its label (W11)
Assure	`status`, `attest`	SESSION call	non-production write	verifiers are principals (A5)
Federate	`export`, `delta`, `reconcile`, `revoke`	CHANNEL of signed view deltas; a SESSION to reconcile	read	a grant per peer and view; labels checked by the exporter

Table 14: The domains of RGP. Each conversation is a connection of Σ_{std}, with an effect class from the lattice of Section 6.4, and each domain is granted separately.

Protocol	What RGP takes	What RGP changes
Chrome DevTools Protocol [332]	domains enabled one at a time; one connection, with a session per target	targets are authorities and hosts; message types are named by contract fingerprints
Language Server Protocol [245]	capabilities negotiated at the start; partial results for long answers	the server knows a system, not one language
Debug Adapter Protocol [244]	lazy expansion: a reference is a promise of detail	references are stable identifiers, not handles that die with the session
Kubernetes watch [339]	list, then watch from a version; an expired version is an error; bookmarks keep idle cursors alive	snapshots and watches are integrals and differences of Z-sets; an expired cursor is reconciled by digests
Reactive Streams [286]	demand signalled by credits	a producer out of credit merges deltas by addition
Git, Dynamo [90, 334]	negotiation and anti-entropy over Merkle digests	the digests are those of the containment forest
OpenTelemetry [275]	resource identity: service, version and instance	observations are joined to elements, not to names
The Update Framework [305]	signed metadata with roles, thresholds, expiry and rotation	patches, claims and exported views are signed, as releases are
DStar, Fabric [226, 364]	labels checked where data leaves a machine; nodes that trust one another differently	the graph's own metadata is labelled, and views are filtered before export
Model Context Protocol [18]	tools for agents	each tool is one RGP operation under the agent's grants

Table 15: What RGP takes from earlier protocols, and what it changes.

monotone traversal needs no coordination. A non-monotone one needs a consistent cut, and without one the answer is UNKNOWN (Proposition 8).

The earlier specification's revision domain needs no messages of its own. Comparing two revisions is the graph diff of Section 9. Asking which live consumers still bind a contract that only an old generation provides is a traversal over runtime facts, and an empty answer is what permits retiring that generation.

12.4 Watching: deltas, cursors and reconciliation

A subscription is a traversal maintained incrementally. The change log already records each commit as a Z-set of inserted and removed rows (Section 11). For a subscribed view $V = \theta(G)$, the producer evaluates the incremental form of Proposition 7 and sends each change of the view as a Z-set delta ΔV. The subscriber integrates: its copy of the view is the sum of the deltas it has applied. Kubernetes' list-then-watch [339] thus becomes DBSP's integration of differences [46]. A snapshot is the integral up to a cursor, and a watch is the stream of differences after it. Four rules keep a watch truthful.

(a) *A delta names the cursor it starts from.* A cursor is a position in one producer's log under one epoch. For definitions it is an authority and a revision; for runtime facts it is a host, an epoch and a sequence number. A delta carries the cursor it starts from and the cursor it ends at. The subscriber applies a delta only if it starts at the subscriber's own cursor. Any other delta marks the subscription stale, and the subscriber asks for a reset.

(b) *The producer never waits, and it merges by addition.* Credits bound what is in flight, as in Reactive Streams [286]. Z-sets form a group, so deltas compose by addition. A producer that runs out of credit stops sending. Later it sends one delta, from the subscriber's cursor to its own head, that is the sum of the deltas it held back. Nothing is lost, and the merged delta is never larger than the change it describes. Keeping only the latest message is correct for a signal of

```
contract  uint    0x3b9e51c4, fingerprint of the Delta schema
conv      uint    7, a watch of routes.reachable at ref main
coverage  map     { excluded: 0, opaque: 1, withheld: 2 }
from      blob    cursor: authority workspace, revision r41
kind      uint    5 (Delta)
rows      vector  Z-set of view rows
  [0]     map     { elem: blob16, value: true, weight: 1 }
  [1]     map     { elem: blob16, value: true, weight: -1 }
to        blob    cursor: authority workspace, revision r42
```

Listing 12: A delta of a watched view as an `rgp/1` frame: a FlexBuffers map with sorted keys.

state, and wrong for deltas, which are differences. Streams that are not views, such as raw samples, may drop under pressure. The next frame then starts at a later cursor, so the gap is visible.

(c) *A cursor includes the epoch.* A host's sequence numbers start again when the host restarts. A cursor from another epoch, or one older than what the log retains [208], is answered with a reset rather than with deltas.

(d) *A reset reconciles.* For a view that is a restriction of the containment forest, the subscriber sends the digests of the scopes it holds. The producer descends the Merkle forest of Definition 4 and returns only the subtrees whose digests differ, as Dynamo's anti-entropy and Git's negotiation do [90, 334]. The cost grows with the size of the difference times the depth of the forest, not with the size of the graph. Any other view is re-evaluated at the head and sent whole.

Listing 28 (Appendix C.1) models the first three rules. Its producer commits, sends single or merged deltas while it has credit and drops them when it has none; the connection can be lost, and the producer can restart under a new epoch. Its invariant, `Truthful`, says that the subscriber's copy is always the producer's state at some cursor of one epoch: possibly old, never wrong. TLC finds that it holds over 2,676 states and that removing any one rule breaks it (Table 34). Without the check of the starting cursor, a dropped delta vanishes in six steps. With cursors that omit the epoch, a subscriber that resumes after a restart mixes two epochs in eleven steps. Merging by keeping the latest delta loses a change in five.

12.5 Changing the graph

The four kinds of change in Section 9 have four authorities, and RGP sends each to its own. Definition commits go to the graph authority, and activations go to hosts. Executors perform data transactions and external effects against stores and providers, and RGP carries only their receipts. Rows never travel in RGP, because rows are not in the graph. Erasing a user's data is a data plan with a receipt from every store (P10 in Section 16), not a message to the authority.

Proposals. A proposal carries a patch, its base revision, its read set and its author (Section 9). The authority answers `check` with typed data:

- the findings of $\mathsf{WF}_{\mathsf{static}}$, computed incrementally by Proposition 9;
- the graph diff, ranked by risk;
- the claims that the patch would make stale;
- the open proposals with which it commutes (Theorem 10).

An approval is a message from a second principal, because no principal approves its own proposal (A5). `commit` moves a ref if the base is still current or no predicate in the read set has changed. Otherwise it fails with the conflicting proposal and a witness, never with a silent last write.

Outcomes. A commit is an effectful SESSION, so its outcome has four cases (W5). Its effect identity binds the digest of the request. A retry with the same identity returns the recorded outcome, and reusing the identity with a different digest is a rejection [219]. A client whose call times out holds UNKNOWN and calls `lookup`. The authority then either returns the recorded outcome or records the identity as refused, so that a delayed copy of the request cannot commit afterwards. Only then may the client conclude that nothing happened. This is the effect-identity template of Table 34, applied to the protocol itself. Listing 13 shows a client.

Edit ownership. The earlier specification made every patch state its write target: a document, source code or a running system. The reason was that the next build erases an edit to structure derived from source. In RG each scope declares its edit owner, so the authority decides instead.

- A patch to a source-owned scope is accepted only as the output of elaboration, and it carries the source revision it was elaborated from as a DERIVES assertion. Any other edit to such a scope goes to a source adapter, which proposes a change to the source under the lens laws of Section 9.
- A graph-owned scope accepts patches directly.
- A running system never changes because of a patch. It changes only when a host admits a release.
- Presentation state (layout, folds and comments) has no invariants. It travels as CRDT operations outside commits [310].

```
val rgp = connect(authority("workspace"), principal = agent("planner"))
val routes = rgp.watch(reachable, at = ref("main"), credit = 64)
launch { routes.deltas.collect { delta -> table += delta } }

val proposal = rgp.propose(addAuth)
val approved = rgp.awaitApproval(proposal, by = role("reviewer"))
val effect = EffectId.fresh()
when (val outcome = rgp.commit(approved, effect)) {
    is Outcome.Success  -> rgp.offer(release(region("chat"), outcome.value))
    is Outcome.Rejected -> report(outcome.findings)
    is Outcome.Failure  -> retryLater(outcome)
    is Outcome.Unknown  -> rgp.lookup(effect)
}
```

Listing 13: A client of RGP that watches a view and commits a patch (proposed API). The traversal is that of Listing 9 and the patch that of Listing 4.

12.6 Releasing to hosts

The release domain carries the protocol of Section 9.5.

- `offer` delivers a signed release with its update-framework metadata [305].
- The host evaluates the admission predicate itself, because admission is decided where it is enforced, and it answers with a generation or with findings.
- `activate` names a mode: next entry, or drain and swap.
- `retire` fences the retired generation's epoch at the stores that own its writes. The stores enforce the fence, and RGP carries their receipts.
- A rollback is a new release under a new epoch.

Each of these is an effectful SESSION with an effect identity, so a host that goes offline during an activation answers `lookup` when it returns. Hosts that are often offline, such as phones, pull: releases wait on a durable channel that replays from an offset. Each host reports the generations it has admitted as runtime facts, so condition (ii) of Definition 10 is observed rather than assumed. The effect class of a release is the join of its contents' effects. A release that contains an irreversible migration is therefore a data migration, and it needs a grant for one.

12.7 Observing and assuring

Observations are joined to the graph by identity (Section 5.4). RGP carries four kinds of runtime fact:

- activations and generations, whose lifecycle arrives as deltas on activation rows rather than as named callbacks;
- the state of live connections: credit, stalls and outcomes, UNKNOWN included;
- receipts of effects and of data plans;
- profiles for the compiler (Section 10.7).

A stalled stream is reported as one. Otherwise a consumer that has stopped granting credit is invisible: no log line records it, and no snapshot shows it. RGP is not a telemetry pipeline. Spans and metrics go through OpenTelemetry to the observation sink [275], and RGP refers to them by identity and carries aggregates and samples.

Payloads and labels. Descriptors are not payloads. A payload is labelled data, and a sample leaves its producer only along a flow that W11 admits: either the reader's clearance dominates the label, or a declassifier on the path holds a grant for the purpose. Redaction is therefore a component in the graph, not a feature of the protocol. The graph's own metadata is labelled too. The names and attributes of a scope can reveal a customer, an incident or a security control, and the graph as a whole is a map of the system (Section 19). Reads, watches and exports filter by the reader's clearance before encoding. A withheld element becomes an opaque stub, which keeps a slice well formed and says only that something is there, and the coverage claim counts such stubs.

Evidence. Verifiers submit evidence as ATTESTS assertions with their artefacts. The status of a claim is a query, and admission reads it (Section 15).

12.8 Federation between authorities

A large system has several graph authorities: partitions of one workspace, separate environments, organisations that share interfaces, and deployments in jurisdictions with their own providers and laws. RGP never merges these authorities and never routes calls between them. Each element has one owning authority, the authority of its consistency domain. Five rules govern the rest.

(F1) *Views, not copies.* An authority exports a view: a restriction along a map of schemas, such as the boundary ports and contracts of the definition layer, filtered by the importer's clearance and signed. Views commute with patches (Proposition 2), so the export of a patch is a patch of the view, and exports are incremental. The importer holds the

view as a replica related to its source by Replicates, with the exporter as the authority. The importer cannot edit the replica, so it never becomes a second source of truth.

(F2) *References pin.* A reference to a foreign element names its authority, its identifier and the revision of the view against which it was checked. When an export removes the element, the importer gets a finding at each reference, a typed hole, rather than a dangling pointer.

(F3) *Calls are declared.* A call between authorities is a Session in both graphs. A gateway realizes it, each side admits the gateway, and each side's grants cover it. Each authority checks the flows that leave it (W11) and the effects that enter it (A5). Comparing environments, such as staging with production, reads two views and routes nothing, so a laptop cannot appear to call production.

(F4) *Digests prove agreement.* Elaboration is deterministic (Section 10), so two authorities that elaborate the same source revision hold equal digests for its definition regions. They can show that they run the same definitions by comparing root digests, without exchanging the definitions, the deployment or anything they have observed.

(F5) *Trust is per authority.* Each authority has its own signing roots and its own cryptographic suites. A peer's root is accepted through a Grant that names the views the peer may read and sets an expiry. A jurisdiction that mandates particular algorithms changes a suite, not the protocol.

Reads across authorities use revision vectors, and a non-monotone read needs a consistent cut across the authorities involved. DStar checks labels where data leaves a machine, and Fabric federates nodes that trust one another to different degrees [226, 364]. RGP applies the same discipline to the graph and its views.

Two jurisdictions. Consider a product that operates in two jurisdictions whose providers and data rules differ, such as the reference stack and the Chinese cloud stack of Section 13.3. It runs one authority in each.

- Both authorities elaborate the same source, so their definition digests agree. Only their Places and Realizes assertions differ (Section 13.1).
- Contracts are exported in both directions, so a session between the two deployments is typed on both sides.
- Runtime facts, observations and profiles stay with the authority where they arise, so profile-guided plans are made per authority (Section 10.7).
- A transfer of personal data between the two is a declared session whose flows and grants each side checks. Claims about its legal basis are part of the evidence that admission requires.

12.9 Carriers, security and conformance

Carriers. RGP needs four properties from a carrier:

- ordered streams with independent flow control, so that a slow subscription cannot block a command;
- credits;
- mutual authentication of principals;
- integrity.

Within a process, RGP runs over an FFI trunk, such as JSI next to a Hermes engine. Between processes and machines it runs over WebSocket or QUIC streams [177]. For hosts that are often offline it runs over a durable channel with replay. A carrier realizes RGP's own sessions and channels, so it is admitted by rule (Real) and condition A2 like any other binding.

Security. Authentication happens at `hello`, and a client without a principal sees nothing. Grants are per domain and per scope, the default is deny, and reads count as effects (A5). The rate and budget attributes of a grant bound what a client may ask for. Agents reach RGP through tools of the Model Context Protocol that map one to one onto its operations (Section 19). An agent's authority is therefore exactly its grants, and labels bound what untrusted input can make it do. Every effectful exchange leaves a receipt that names the principal, the effect identity, the request digest and the outcome.

Conformance. The RGL Core conformance suite (Section 10.8) covers RGP with frame vectors for every message type, transcripts of conversations with their expected outcomes, and the TLA$^+$ templates of Table 34. A conforming endpoint:

- names elements only by identifier;
- answers every read with coverage, and evaluates traversals before encoding;
- never blocks a runtime on a subscriber;
- starts every delta at a cursor, and resets on a mismatch;
- never decodes by position across fingerprints;
- refuses patches to source-owned scopes that elaboration did not produce;
- answers UNKNOWN rather than guessing, and refuses self-approval;
- filters by label before encoding, and never routes between authorities.

One test checks the identity model with two builds and no infrastructure: builds with identical contracts and different bodies must be reported as compatible and distinct.

What RGP is not. RGP is not a telemetry pipeline, a data API, a transport for application sessions or a consensus protocol. Application calls use their own realizations, and RGP carries their descriptions and observations. Rows stay in stores. Each authority commits its own domain, and a change that spans authorities is a plan executed with receipts

Capability class (used by)	Required guarantee	Reference realization	Other realizations
Stateless compute place (components, Session servers)	Runs Kotlin/JS or JVM code; binds resources; starts in milliseconds	Workers: V8 isolates [68]	Lambda, Cloud Run, Kubernetes pods
Keyed single-writer place (exclusive profile, keyed Realizes)	One live instance per key; serialized turns; transactional storage co-located with the instance	Durable Objects with SQLite storage [64]	Orleans-style actor hosts; a store-backed actor host with fencing
Transactional store (`atomic(store)`, row policies)	ACID transactions; row-level security or equivalent; a change feed	Postgres, managed by Supabase [324], reached through Hyperdrive	Aurora, RDS or Cloud SQL Postgres; self-hosted Postgres
Durable channel (queues, topics)	At-least-once delivery, retries, a dead-letter target; per-key order where required	Cloudflare Queues [66]; Pub/Sub with ordering keys [153]	SQS (FIFO), Kafka [208], NATS JetStream
Durable execution (durable Transitions, timers, human waits)	Persisted step outcomes, retries, timers, external events	Cloudflare Workflows [69]; Durable Object alarms	Temporal [329], Step Functions
Object storage (artefacts, blobs, releases)	Durable, content-addressed names	R2	S3, GCS, Ceph
Long-running service place (JVM services, engines)	Containers with persistent volumes	k3s [190] on Compute Engine	Any Kubernetes; virtual machines
Identity and secrets (Principal, Grant)	Token verification at the edge with key rotation; a secret store	ES256 tokens verified against JWKS; Workers secrets; Secret Manager	Keycloak, Vault, cloud KMS
Observation sink (Observation)	Trace ingestion with typed attributes and retention	OpenTelemetry [275] into ClickHouse on k3s	Any OpenTelemetry backend
Analytical store (OLAP over events, observations, features)	Columnar storage; fast aggregation over append-only data; materialized views	ClickHouse [63, 308] on k3s	BigQuery, Druid, DuckDB
Graph store (graph queries over derived graphs)	A property graph with Cypher and in-memory traversal; a change feed	Memgraph [239] on k3s	Neo4j, Neptune, recursive SQL
Device and browser places (Surfaces, pushed regions)	ART, Kotlin/Native, a JavaScript engine, Hermes	Android, iOS, browsers, Hermes engines	The same

Table 16: The capability profile of an RG host, its reference realization and alternatives. A placement is admissible only through qualified realizations (A2).

(Section 9).

13 A reference realization

RG is provider-neutral. A host qualifies for a placement when it offers the capability classes of Table 16 through realizations that are qualified for the families they carry (A2). We nonetheless commit to one realization and build it first. Orleans was designed alongside one cloud and proved there while remaining portable [34]; a concrete target forces the semantics to meet real limits, and it is where the measurements of Section 20.4 will be taken. Reaktor, as released, consists of RG and RGL, the modules and tools of Section 14, and the realization described here:

- Kotlin on every runtime;
- Cloudflare for compute, keyed state, messaging and storage at the edge;
- Google Cloud for long-running services and cross-cloud messaging;
- Postgres as the transactional authority.

In our assessment it is a good default for a typical multi-runtime application, not for every one. Section 13.3 gives four further stacks: the reference stack at large scale, a Chinese cloud stack, an alternate public cloud stack and a corporate stack hosted internally.

13.1 What a host must provide

Composing missing classes. A provider that lacks a class can still host RG if the class is built from others and the composite is qualified. A keyed single-writer place, for example, can be built from a transactional store plus epoch fencing; the handover model of Table 9 states exactly what that realization must enforce. Admission refuses a placement whose realizations are unqualified, so portability is checked rather than assumed. Where the reference providers are unavailable, a deployment changes only its Realizes and Places assertions; scopes, contracts and patches stay the same (T2). That covers an on-premises cluster, an air-gapped site, or a jurisdiction served by other providers; Section 13.3 works through such cases.

13.2 Why this stack

Figure 2 places a whole platform on this stack, from devices to claims.

Edge isolates as the default place. Workers run V8 isolates inside Cloudflare's network. They start in milliseconds, are billed by use and call one another through service bindings without leaving the network [68]. This suits the common case: interactive, read-heavy, globally distributed traffic with bursty writes.

Durable Objects as the actor primitive. A Durable Object is a single instance per identifier, with its own transactional storage [64]. That is the exclusive profile with keyed realization, supplied by the platform. Uniqueness and placement come from the provider; RG adds typed protocols, supervision, effect identity, and fencing at the external stores that check epochs.

- *Turns.* Input gates defer other events only while storage operations are pending, so an await on the network lets another event in. The exclusive profile's one-message-per-turn law is therefore realized by an explicit turn loop over the mailbox, which the qualification suite tests.
- *Idle connections.* WebSocket hibernation lets a room keep idle connections without keeping the object resident.

Messaging and durable execution next to the compute. Queues deliver at least once, with retries and dead-letter queues, but order is only best effort [66]. They therefore qualify for competing-consumer channels, not for ordered ones (Section 7). Workflows persist each step's result, retry steps, sleep, and wait for external events with timeouts [69]. They realize durable transitions, including human gates.

Postgres as the one transactional authority. Postgres is a capable, general-purpose transactional store, and most providers offer it, so it is easy to move. Row-level security lets grants compile to row policies, and the outbox commits in the same transaction as the effect. We use a managed provider for Postgres only, so the authority stays portable.

Google Cloud for what isolates do poorly. Long-running JVM services and two engines run on k3s on Compute Engine; Workers reach the cluster over a private tunnel. ClickHouse holds events, observations and features for analytics: its columnar storage aggregates billions of rows, and observations joined by element identifier become queries rather than log searches. Memgraph holds graphs derived from the data, such as the follow graph, and projections of the system graph itself, for queries that are natural in Cypher: neighbourhoods, paths and communities. Both are stores in RG terms: their datasets are related to authoritative ones by Derives, and traversals compile into their query languages (Section 8). Pub/Sub carries cross-cloud fan-out with ordering keys, and offers exactly-once delivery to pull subscribers within one region [153].

Where this stack is not the best choice. RG then places the affected components on other places; the graph does not change. The cases are:

- heavy batch and GPU training;
- very high write throughput to one region;
- state that outgrows a single object;
- fast-paced games that need unreliable datagram transport;
- jurisdictions without these providers (Section 13.3).

13.3 Four further stacks

The reference stack is one profile among several. Four others recur often enough to state here:

- the reference stack at large scale;
- a Chinese cloud stack, for deployments in mainland China;
- an alternate public cloud stack, for organisations committed to one hyperscaler, with AWS as the example;
- a corporate stack hosted internally, on premises or in a private cloud, sometimes without access to the internet.

Each stack chooses a realization for every capability class of Table 16, and Table 17 lists the choices. Moving a system between stacks changes four things:

- its Places and Realizes assertions;
- the modules that wrap provider services, such as push, sign-in, payments and model hosts;
- the generated deployment layer;
- the evidence of qualification.

Scopes, contracts, patches and behaviour stay the same (T2). The substantial work is qualification. A class that a stack lacks is composed from others and qualified as a composite (Section 13.1). None of the four stacks is qualified yet. The plan qualifies the reference stack first (Table 30), and each further stack is qualified by running the same suites against its providers.

The reference stack at large scale. Scale changes the number of authorities, not their kinds.

- *Authorities are partitioned.* Stores are split by tenant or region. Each part is a consistency domain with its own primary and read replicas, and a sharding layer such as Citus [78] serves a tenant that outgrows one primary. A change that spans partitions is a saga with receipts, never a distributed transaction (Section 9). The graph authority is partitioned in the same way, and its partitions federate through RGP (Section 12.8).

Capability class	Reference stack at large scale	Chinese cloud	Alternate public cloud (AWS)	Corporate, hosted internally
Stateless compute	Workers in every region; Smart Placement near the data	Function Compute, SCF or FunctionGraph; ESA Edge Routines or EdgeOne edge functions	Lambda, with Node.js for Kotlin/JS and SnapStart for the JVM; Lambda@Edge	pods on Kubernetes; workerd for code written for isolates [70]
Keyed single-writer place	Durable Objects, with hot keys split; namespaces restricted to a jurisdiction [65]	composed: an actor host on ACK, TKE or CCE, fenced by the store	composed: an actor host on EKS, fenced by DynamoDB conditional writes or in Postgres	composed: an actor host on Kubernetes, fenced in Postgres; Orleans or Akka where already used
Transactional store	Postgres primaries per tenant or region, with read replicas; Citus for the largest tenants [78]	RDS or PolarDB for PostgreSQL; TencentDB for PostgreSQL	Aurora PostgreSQL; Limitless Database for sharding [14]	Postgres run by CloudNativePG or Patroni
Durable channel	Queues partitioned by key, each under its limit of 5,000 messages per second [67]; Pub/Sub or Kafka beyond	RocketMQ (ApsaraMQ or TDMQ) with ordered messages and dead letters; Pulsar	SQS, FIFO for order per group; SNS for fan-out; Kinesis or MSK for logs	Kafka with Strimzi, NATS JetStream or RabbitMQ streams
Durable execution	Workflows; Temporal on GKE for long, high-volume flows	Temporal on Kubernetes, or CloudFlow once qualified	Step Functions; Temporal on EKS	Temporal [329]
Object storage	R2, with Cloud Storage next to the clusters	OSS, COS or OBS	S3	Ceph's object gateway or SeaweedFS
Long-running services	regional GKE clusters	ACK, TKE or CCE	EKS or ECS	OpenShift, Rancher or upstream Kubernetes
Identity and secrets	as in Table 16, with keys per region	sign-in by phone number, WeChat or Alipay; the provider's KMS; SM2, SM3 and SM4 where required	Cognito or an external identity provider; KMS and Secrets Manager	the corporate directory over OIDC or SAML, brokered by Keycloak; Vault or OpenBao
Observation sink	collectors per region with tail sampling, into sharded ClickHouse	OpenTelemetry into ClickHouse, inside the country	the AWS Distro for OpenTelemetry into ClickHouse on EKS	OpenTelemetry into ClickHouse; Grafana
Analytical store	ClickHouse shards and replicas, coordinated by ClickHouse Keeper	ApsaraDB for ClickHouse or TCHouse-C; Doris or StarRocks	ClickHouse Cloud or ClickHouse on EKS; Redshift	ClickHouse
Graph store	Memgraph with replicas; the largest derived graphs as recursive SQL	TuGraph with openCypher, or NebulaGraph with nGQL	Neptune with openCypher	Memgraph or Neo4j
Devices and browsers	as in Table 16; releases staged by cohort	Android through vendors' stores, without Google's services; iOS; HarmonyOS NEXT; WeChat and Alipay mini-programs; H5 pages	as in Table 16	managed devices; enterprise distribution
Model hosts	provider APIs behind Sessions, with fallbacks	domestic models on provider platforms, or open weights on GPU nodes	Bedrock, or open weights on GPU instances	open weights on internal GPU nodes, served with vLLM [209]

Table 17: Four further stacks, by the capability classes of Table 16, with model hosts added. A class marked *composed* has no direct counterpart on that stack; it is built from other classes and qualified as a composite (Section 13.1). The products named are realizations to be qualified; none of the four stacks is qualified yet.

- *Hot keys are split.* A Durable Object serializes the turns of its key. A key with more traffic than one object can serve, such as a very large room, is split into several keys. The split is a patch to the keyed realization, and observations identify the hot keys (Section 10.7). Namespaces restricted to a jurisdiction keep objects and their storage in the European Union, in FedRAMP data centres or in the United States [65].
- *Heavy streams leave the edge queue.* A Cloudflare queue accepts up to 5,000 messages per second [67]. Streams beyond that are partitioned over several queues, or moved to Pub/Sub or Kafka. Code sees the channel's quality of service, not its realization.
- *Clusters become managed and regional.* Regional GKE clusters replace k3s. ClickHouse runs as a cluster of shards and replicas coordinated by ClickHouse Keeper. Memgraph keeps replicas, and a derived graph that outgrows memory is partitioned, or its traversals are compiled to recursive SQL instead (Section 8.3).
- *Observation is sampled.* Collectors in each region sample traces by their tails, keeping errors and slow paths in full and the rest in proportion. Coverage claims state the sampling (Definition 8).
- *Releases are staged.* A release reaches cohorts of hosts in turn, so generations coexist for longer, and the compatibility checks for version skew (P13 in Section 16) carry more of the load.

At this scale, cost joins latency among the preferences of the placement solver (Section 6.6).

A Chinese cloud stack. In mainland China the reference providers are unavailable or reachable only through local partners. Personal information is subject to rules on where it is stored and when it may leave the country [318, 319]. The stack therefore uses Alibaba Cloud, Tencent Cloud or Huawei Cloud. Their services cover every class of Table 16 except the keyed single-writer place, which is composed from an actor host and a store that fences by epoch. Six further differences matter.

- *A separate authority.* The deployment runs its own graph authority inside the country. It elaborates the same source as the authority abroad, so the two hold equal definition digests, and they exchange contracts as signed views. Runtime facts, observations and profiles stay in the country (Section 12.8).
- *Devices.* Android phones are reached through their vendors' stores, without Google's services, so push notifications

go through the vendors' own channels as a module realization. HarmonyOS NEXT is a further device place. Its applications are written in ArkTS, a language based on TypeScript, and they reach native code through Node-API in C, so the glue languages of Section 13.5 cover it. Kotlin reaches it through Kotlin Multiplatform frameworks such as Tencent's Kuikly [330], not yet through an official Kotlin target. Mini-programs inside WeChat and Alipay are restricted JavaScript places. They are released through the host application's review, so a change reaches them as a new mini-program release, not through RG's code push.

- *Identity and payments.* Sign-in by phone number, WeChat or Alipay, and payment through the latter two, are module realizations behind the same contracts.
- *Cryptography.* Sectors that mandate the national algorithms SM2, SM3 and SM4 change the suites of RGP and of the stores, not the protocol (F5 in Section 12.8).
- *Models.* Domestic language models, served by the providers' platforms or as open weights on GPU nodes, realize the same Sessions as any other model host.
- *Obligations outside the graph.* Regulators decide filings for sites, applications and generative services [79], and grade systems under the multi-level protection scheme [317]. The graph cannot discharge these obligations, but it supplies their evidence as claims (Section 15): the inventory of labelled data, the flows that leave the country, and where each store is placed.

An alternate public cloud stack. An organisation committed to one hyperscaler places everything there. With AWS as the example, three differences from the reference stack matter.

- *No keyed single-writer place.* AWS has no counterpart of a Durable Object. Exclusive components therefore run on an actor host on EKS, and their writes are fenced in the store, by a conditional write in DynamoDB or an epoch check in Postgres. The handover model of Table 34 is the specification such a host must meet.
- *Less at the edge.* Edge functions do less there, so more components are placed in a region, and latency budgets account for the extra distance (Section 3).
- *Managed sharding of Postgres.* Aurora PostgreSQL Limitless Database distributes tables over several writers behind routers [14]. Whether a transaction across its shards is one consistency domain in RG's sense is a property for its qualification to establish, not to assume.

The other hyperscalers fit the same table. On Azure, Durable Entities and Orleans [34] provide the keyed single-writer place directly, and Service Bus sessions give order per key. Google Cloud alone covers the classes with Cloud Run, Cloud SQL or AlloyDB for PostgreSQL, Pub/Sub, Workflows and GKE, again with a composed keyed place.

A corporate stack hosted internally. An enterprise that keeps code, data and the graph inside its own network runs every class on its own Kubernetes clusters.

- *No edge network.* Stateless components run in pods, or in workerd, the open-source runtime of Workers [70], which runs the same Kotlin/JS modules; their bindings to managed services are realizations, and change with the stack. Keyed places are composed, as on AWS.
- *The control plane is the customer's.* The graph authority, RGP and the observation stores run inside the network. A vendor's authority sees only the views that the customer exports, or nothing.
- *Identity comes from the directory.* The corporate identity provider issues the principals over OIDC or SAML, and directory groups map to roles that hold grants. Approvals follow the organisation's own separation of duties (A5).
- *Sites without the internet.* An air-gapped site receives releases, models and updates of RG itself as signed artefacts carried in by hand. Their update-framework metadata [305] lets hosts verify them without reaching the signer, and the durable release channel replays them in order (Section 12.6).
- *Managed devices.* Applications reach managed devices through enterprise distribution, so the distribution policy in the admission predicate (Section 9.5) is the organisation's as well as the platform's.
- *Models stay inside.* Open-weight models are served on internal GPU nodes, for example with vLLM [209], and an external model is used only where a declared flow and a grant allow it (W11).

Several stacks at once. Stacks combine. A product can serve most users on the reference stack and one jurisdiction on the Chinese stack. An enterprise customer can run the corporate stack and federate with the vendor's authority. In both cases the placement solver sees the union of the host profiles, Places constraints keep each store and activation inside its stack, and the authorities federate through views (Section 12.8). Contracts are shared and checked on both sides; nothing else crosses unless a declared flow allows it.

13.4 Kotlin as the implementation

Kotlin Multiplatform compiles the same common code to four targets [183]:

- the JVM, for cluster services;
- JavaScript ES modules, for Workers, Durable Objects, browsers and Hermes engines;
- native code, for iOS;
- Android.

```
class Activation(spec: ActivationSpec, parent: CoroutineScope, place: Place) : CoroutineScope {
    override val coroutineContext: CoroutineContext =
        parent.coroutineContext + supervision(spec, parent.coroutineContext[Job]) +
            place.dispatcher + ActivationKey(spec.key, spec.epoch)
}

fun supervision(spec: ActivationSpec, parent: Job?): Job = when (spec.supervision) {
    OneForOne -> SupervisorJob(parent)
    AllForOne -> Job(parent)
}

suspend fun <R, E> SessionPort<R, E>.call(request: Request, deadline: Duration): Outcome<R, E> =
    withTimeoutOrNull(deadline) { send(request) } ?: Outcome.Unknown(request.effectId)
```

Listing 14: Activations and calls on Kotlin coroutines (proposed reference implementation, abridged).

With per-file module output, each Worker bundles only the code reachable from its entry point.

The compiler. RGL uses the Kotlin compiler in three places [180, 181]:

- *Symbol processing* reads declarations. It extracts contracts and their serializers (hence fingerprints), registries and FlexBuffers codecs.
- *Frontend checkers* in a K2 compiler plugin report RG's static rules as ordinary diagnostics in the IDE:
  - capture analysis for code entries;
  - BIND locality when placement is static;
  - ownership of borrowed and moved values (W9);
  - effect ceilings (W10);
  - label flow on statically known paths (W11);
  - contract compatibility (W5).
- *The plugin's backend extension* performs four transformations:
  - it lifts code entries out of closures into registered entries;
  - it stamps stable identifiers derived from declaration paths, not object identity;
  - it inserts interceptor and observation hooks only where a plan requires them;
  - it records source spans, so every element can be traced to the code that defines it.

The Compose compiler and kotlinx.serialization are the precedents [149, 184]: each delivers a programming model through a compiler plugin while the source stays ordinary Kotlin.

Coroutines. The runtime containment forest is the coroutine job tree (Section 8.4) [105].

- *Scopes.* An activation's scope is a child of its parent activation's scope.
- *Supervision.* A one-for-one supervision facet is a supervisor job, all-for-one is an ordinary job, and escalation is an uncaught failure that propagates to the parent. Deactivation is cancellation.
- *Dispatchers* are capabilities of the place:
  - the main thread on Android;
  - the event loop in a Worker;
  - the microtask queue in Hermes;
  - a bounded pool on the JVM.
- *Families.*
  - A SESSION call is a suspending function that returns an Outcome. Its deadline is a timeout whose expiry yields UNKNOWN, never failure.
  - A signal CHANNEL is a state flow and an event channel is a flow.
  - A mailbox is a coroutine channel whose capacity and overflow are the channel's declared attributes.
  - A choice in a multiparty session is a `select`.
- *Durability.* Durable work never serializes a coroutine stack: a durable component persists each step's outcome and resumes by replaying outcomes.

Listing 14 shows the core of the mapping.

Code push. The Kotlin/JS backend emits an ES module that `hermesc` compiles to bytecode; the same module runs in a Worker. A pushed region's host imports are exactly its capability ports (Section 10.6), and its render plans may use only the surfaces its host offers (Section 8.3).

13.5 Beyond Kotlin: languages, formats and checkers

Kotlin is the implementation language. Five other technologies have fixed parts: C++ for native components, FlexBuffers at every boundary, and TLA$^+$, Alloy and Lean as checkers. Each acts at a definite place and in a definite layer, and each

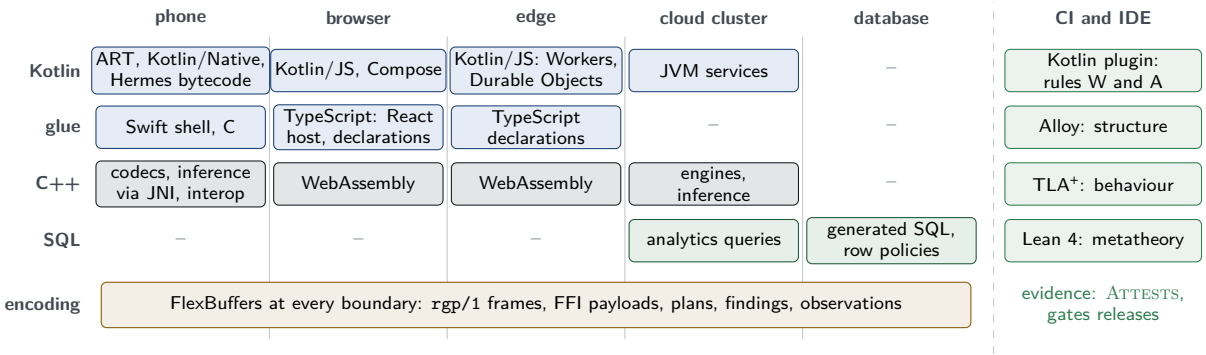


Figure 21: Where each technology acts. Rows are languages, the glue languages and the encoding; columns are the places of the reference stack and, on the right, the checkers that run in CI and the IDE. A dash means the technology has no part at that place.

Technology	Where it acts	What it realizes or checks	How it meets the graph
Swift, C, TypeScript	the iOS shell, native interfaces, JavaScript hosts	glue only; none of Reaktor's semantics	generated bindings and declarations
C++	phones, browsers (Wasm), servers, robots	native components: codecs, inference engines, physics, control loops; a reader of graph slices	FFI trunks with ownership modes; contracts generated as C++ types
FlexBuffers	every boundary	the encoding of `rgp/1` frames, FFI payloads, plans, findings and observations	read in place; sorted keys give stable bytes; a fingerprint names the schema
TLA$^+$	CI and design review	behaviour: family templates, releases, trace validation	templates instantiated per region (Equation (1)); results recorded as ATTESTS evidence
Alloy	IDE and CI	structure: the static rules over the C-set schema; changes to the signature	schema and rules as signatures and facts; counterexamples become findings
Lean 4	proof repository	metatheory: preservation, determinism, compositionality	definitions that mirror the schema; theorems cited by claims

Table 18: Technologies beside Kotlin in the reference realization.

meets the graph through a typed interface rather than by convention. Three glue languages, Swift, C and TypeScript, only connect Kotlin to platforms and ecosystems. Figure 21 shows where each one acts, and Table 18 summarises what it realizes or checks.

Glue languages. Swift, C and TypeScript appear only at boundaries, and none of them implements Reaktor's semantics. Swift hosts the iOS shell and calls into Kotlin/Native. C is the binary interface between Kotlin/Native, Swift and C++. TypeScript supplies and receives declarations: karakum turns npm and Cloudflare type declarations into Kotlin bindings, and Kotlin/JS exports declarations for the React page that hosts Compose on the web and for any component a third party writes in TypeScript behind typed ports.

C++. C++ runs where Kotlin would be too slow or where the code already exists: media codecs, inference engines, physics and robot control. A C++ component sits behind an FFI trunk. That is JNI on Android, Kotlin/Native interoperation on iOS, JSI next to a Hermes engine and WebAssembly in a browser or a Worker; on a server it is a separate process reached by a SESSION. Ownership modes become C++ types: move is a unique pointer, borrow a span valid for the duration of the call, and share a shared, read-only lease. The trunk checks them at the boundary, so a borrowed camera frame cannot outlive the call that lent it (W9). Listing 15 shows the media codec of Figure 2. For hosts that have no Kotlin runtime, such as robots or game engines, a C++ reader of the kernel's column dumps and FlexBuffers frames lets them load plans and graph slices without a JVM.

FlexBuffers. Every boundary in Figure 2 carries FlexBuffers [148]: the `rgp/1` graph protocol (Section 12), FFI payloads, plans, findings and observation batches. Four properties decide the choice.

- A receiver reads a frame in place, from a JVM byte buffer, a JavaScript array buffer or C++ memory, without parsing it into objects.
- Frames describe themselves, so developer tools and logs can print any frame without generated code. The contract fingerprint in the frame names the schema that the receiver checks.
- Maps keep their keys sorted, so encoding a canonical form yields stable bytes for digests and signatures.
- Frames are compact, and vectors of integers serve as column chunks.

```
class MediaApi {
public:
    virtual Outcome<EncodedFrame, MediaError> encode(std::unique_ptr<Frame> frame,
                                                     EncodeOptions const& options) = 0;
    virtual void preview(std::span<std::byte const> frame) = 0;
    virtual ~MediaApi() = default;
};

class MediaCodec final : public MediaApi {
public:
    Outcome<EncodedFrame, MediaError> encode(std::unique_ptr<Frame> frame,
                                             EncodeOptions const& options) override;
    void preview(std::span<std::byte const> frame) override;
};

RG_COMPONENT(MediaCodec, "media-codec", RG_PROVIDES(MediaApi, "MediaApi", "v2", "c41b07d2"))
```

Listing 15: A native component behind an FFI trunk; ownership modes as C++ types (proposed SDK).

```
author    string  "agent:reviewer"
base      blob    32 bytes: digest of revision r41
contract  uint    0x5e1f02a7, fingerprint of the Patch schema
kind      uint    2 (Patch)
ops       vector
  [0]     map     { edge: blob16, op: 1 (AddIncidence), ord: 0, port: blob16, role: 3 }
  [1]     map     { elem: blob16, expect: 7 days, facet: 12, field: 2, op: 4 (SetAttribute), to: 30 days }
sig       blob    64 bytes: Ed25519 over the canonical bytes of every other key
```

Listing 16: The patch of Listing 4 as an `rgp/1` frame: a FlexBuffers map with sorted keys.

Listing 16 shows the patch of Listing 4 as a frame.

TLA$^+$. TLA$^+$ checks behaviour. The five templates of Table 34 are hand-written today. The plan instantiates them per region from the family templates by Equation (1), so that a release ships with the model of its own protocols. TLA$^+$ also has a second use: observations carry element identifiers and epochs, so a recorded trace can be checked against the specification. That tests whether a realization refines its family's protocol in production, not only in the qualification suite.

Alloy. Alloy checks structure [178]. The C-set schema of Figure 6 reads directly as Alloy signatures, and the static rules as facts. Alloy can find an instance that satisfies every rule, which shows that the rules are consistent. It can also find a counterexample to a proposed change of the signature before the change ships. Listing 17 shows the core; it is a sketch that has not yet been run.

Lean 4. Lean proves what a bounded checker cannot: properties of graphs of every size [85]. The definitions mirror the C-set schema, and the theorems are the metatheory of Section 15: preservation of well-formedness by each patch operation (Proposition 9), determinism of traversals (Theorem 6) and compositionality (Theorem 3). Listing 18 states preservation; its proof is planned for Q3 2027 (Table 30).

```
abstract sig Element { owner: lone Element }
sig Node extends Element {}
sig Port extends Element { family: one Family, role: one Role }
sig Hyperedge extends Element { kind: one Family }
enum Family { Bind, Session, Channel, Transition }
sig Role {}
sig Incidence { edge: one Hyperedge, port: one Port, role: one Role }

fact Forest { no e: Element | e in e.^owner }
fact PortsBelongToNodes { all p: Port | p.owner in Node }
fact Nets { all p: Port | lone port.p }
fact Typed { all i: Incidence | i.port.family = i.edge.kind and i.port.role = i.role }

assert OneConnectionPerPort { all p: Port | lone (port.p).edge }
check OneConnectionPerPort for 6
```

Listing 17: The core of the schema and three static rules in Alloy (sketch, not yet run).

```
structure Graph where
  Node : Type
  Port : Type
  Edge : Type
  owner : Port -> Node
  family : Edge -> Family
  role : Port -> Role
  ends : Edge -> List Port

def Graph.nets (g : Graph) : Prop :=
  forall (p : g.Port) (e f : g.Edge), List.Mem p (g.ends e) -> List.Mem p (g.ends f) -> e = f

def Graph.wellFormed (g : Graph) : Prop :=
  g.nets /\ forall (e : g.Edge) (p : g.Port), List.Mem p (g.ends e) -> allowed (g.family e) (g.role p)

structure Patch where
  pre : Graph -> Prop
  apply : Graph -> Graph

theorem preservation (op : Patch) (g : Graph) (wf : g.wellFormed) (ok : op.pre g) :
    (op.apply g).wellFormed := by
  sorry
```

Listing 18: Preservation of well-formedness, stated in Lean 4 (sketch; the proof is planned).

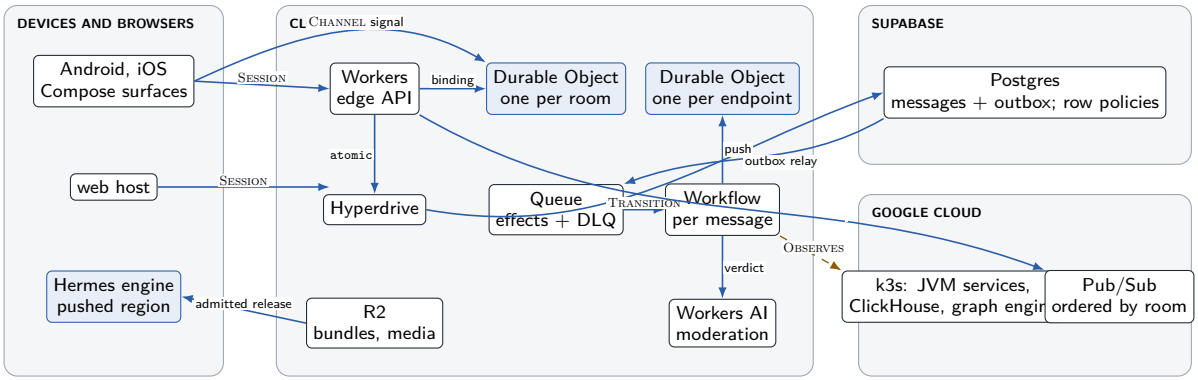


Figure 22: The chat scope placed on the reference stack (Table 19). Keyed exclusive components become Durable Objects; the transactional authority is Postgres, with its outbox drained into a queue; a Workflow orders moderation before the push. Observations and ordered analytics leave for Google Cloud. Every arrow is a realization of a family in the graph, not a hand-written configuration.

13.6 A worked example: the chat scope

Table 19 realizes the scope of Listings 7 and 8 on the reference stack. Figure 1 placed the actor on the phone; here placement puts it in a Durable Object, next to the pool that reaches its store. The definition is unchanged, and only the PLACES and REALIZES assertions differ. Every row is a REALIZES or PLACES assertion produced by placement; none is written by the feature's author. Figure 22 draws the result, and Listing 19 shows part of the Workers configuration that the deployment layer would generate for it, written out by hand.

Four failures show the guarantees composing.

(i) *A Worker crashes after the Postgres commit but before enqueueing the push.* The outbox row survives and the relay sends it, so nothing is lost.

(ii) *The queue redelivers a message.* The push step is idempotent by message id, so nothing is duplicated.

(iii) *A phone's call times out.* The outcome is UNKNOWN, and the client asks by idempotency key instead of resending blindly (Table 9).

(iv) *A room's object is evicted mid-conversation.* Its state reloads from its storage, sequence numbers let clients resume the signal channel, and a client retry carries the same effect id.

None of these four behaviours is written by the feature's author. Listings 7 and 8, about thirty lines, define the feature; moderation and push come from platform scopes that it imports. The realizations are elaborated from them, and a change to any realization is a graph patch, reviewed as one.

```
{
  "name": "chat-edge",
  "main": "chat-edge.mjs",
  "durable_objects": {
    "bindings": [
      { "name": "CHAT_ROOM", "class_name": "ChatActor" },
      { "name": "PUSH_ENDPOINT", "class_name": "PushEndpoint" }
    ]
  },
  "migrations": [ { "tag": "r42", "new_sqlite_classes": ["ChatActor", "PushEndpoint"] } ],
  "queues": {
    "producers": [ { "binding": "CHAT_EFFECTS", "queue": "chat-effects" } ],
    "consumers": [ { "queue": "chat-effects", "max_retries": 5, "dead_letter_queue": "chat-effects-dlq" } ]
  },
  "workflows": [ { "binding": "MESSAGE_FLOW", "name": "message-flow", "class_name": "MessageFlow" } ],
  "hyperdrive": [ { "binding": "CHAT_DB", "id": "generated-at-deploy" } ],
  "services": [ { "binding": "MESSAGING", "service": "messaging" } ]
}
```

Listing 19: The Workers configuration that the deployment layer would generate from the chat scope (excerpt, written by hand). Names, classes, queues and bindings all come from the graph.

14 Reaktor: modules and tools around the graph

Reaktor, as released, consists of four parts:

- RG and RGL;
- the modules that give each kind and family a working implementation;
- the tools through which people and agents read and change the graph;
- the cloud realization of Section 13.

Reaktor today has about forty modules. Table 20 groups them and maps every module and tool to a fragment of Σ_{std} plus traversals and executors. In this mapping none needs a new base kind or a kernel change; H1 (Section 20.4) tests that claim on real graphs. The status column is honest: several modules are partial, some exist only as designs, and their mapping is what the plan in Section 22 implements. Section 14.1 then takes the groups one at a time and states how each module is built on the graph, and Section 14.2 shows what a builder gains when all of them share it.

Evidence of fragmentation. The survey behind this paper (Section 20.2) found the same model reimplemented many times across systems built on the current runtime.

- *A hidden signature.* A knowledge-and-work system keeps about fourteen graph representations. Its registry of 67 typed record collections, composed from 26 domain modules with owners, references, transition rules, authority and a privacy lattice, is a kind signature in all but name.
- *A hand-built change pipeline.* The same system's write path is the change pipeline built by hand: commands with expected revisions, receipts, checks, one gateway transaction, a monotone change log, client outboxes and propose-only agents.
- *Seven models of one system.* The developer workbench holds seven models of the system it inspects, joined by names.
- *Seven topology sources.* A product declares its deployment topology in seven hand-kept places that disagree.

RG removes the reason for each copy. The signature is data, the pipeline is the kernel's, and routes, documents and infrastructure are one graph with projections.

14.1 Each module on the graph

Kernel and language. Today `core` keeps a process-wide registry of slots, one per platform service: permissions, databases, files, the theme, authentication, secrets, haptics, health data, steps, location, eight media services, notifications, telemetry, Pub/Sub and work. These are the 31 process-global service slots of Table 28. `graph-runtime` wires consumers to providers by type and key and falls back to a dependency-injection container. On the graph each slot becomes a capability port that a place offers, resolved with its fallbacks by the support traversal (Section 6.6), and the wiring becomes the lattice traversal of Table 8, which reports a missing or ambiguous provider with a witness instead of failing at run time. `graph-port`'s definition IR already has nested regions, typed ports with contracts and polarity, and hyperedges with ordered, role-labelled incidences; it becomes the kernel's tables almost unchanged. `core` also records the provenance of facts as live, source, inferred, fixture, stale, failed or unknown, which maps onto the origin and freshness dimensions of claim status (Section 15). The codec generator becomes the first stage of the compilation pipeline (Section 10.7), `ffi`'s registry of invokable functions becomes a trunk whose imports are capability ports, and the outputs of the build plugins become artefacts that derive from their sources.

Element	Realization	Why it is admissible
`api`: SESSION call	Phone to Worker over HTTPS with a token; Worker to Worker over a service binding	Deadline and idempotency key carried; a timeout yields UNKNOWN
`actor`: exclusive component keyed by room	One Durable Object per room; SQLite storage as the actor store; an explicit turn loop	One instance per key from the platform; turns tested
`timeline`: signal CHANNEL	Hibernatable WebSockets from the room's object, with sequence numbers for replay from its storage	A signal needs latest-value semantics only
`inbox`: mailbox CHANNEL	The object's request stream into its turn loop	One consumer by construction
`chat-db`: transactional Store	Postgres through Hyperdrive; `atomic(chatDb)` writes the message and an outbox row; row policies generated from grants	One authority; the outbox commits with the effect
Outbox relay	A Worker that drains the outbox into a queue of effects	At least once with dead letters; consumers idempotent by message id
Moderation, then push	A Workflow per message: moderate, guard on the verdict, push, update the room; compensation on delete	Steps persisted; push only after the verdict
Notifications	A Durable Object per device endpoint that records each delivery intent before calling the provider	No duplicate push after a retry
Observations	OpenTelemetry spans carrying element ids and the revision digest, exported to ClickHouse; analytics on a Pub/Sub topic ordered by room	Joins by identity; per-room order where needed
`list`, `chat`: Surfaces	Compose on Android and iOS and a web host; the list surface's logic pushed as Hermes bytecode from R2 after admission	Render plans use host surfaces only
Deployment	A Workers configuration generated by the proposed deployment layer (object namespaces, queues, bindings, Hyperdrive), a Pulumi program for Google Cloud and Postgres migrations	One source; drift is desired ⊖ observed

Table 19: The chat scope of Listing 7 realized on the reference stack.

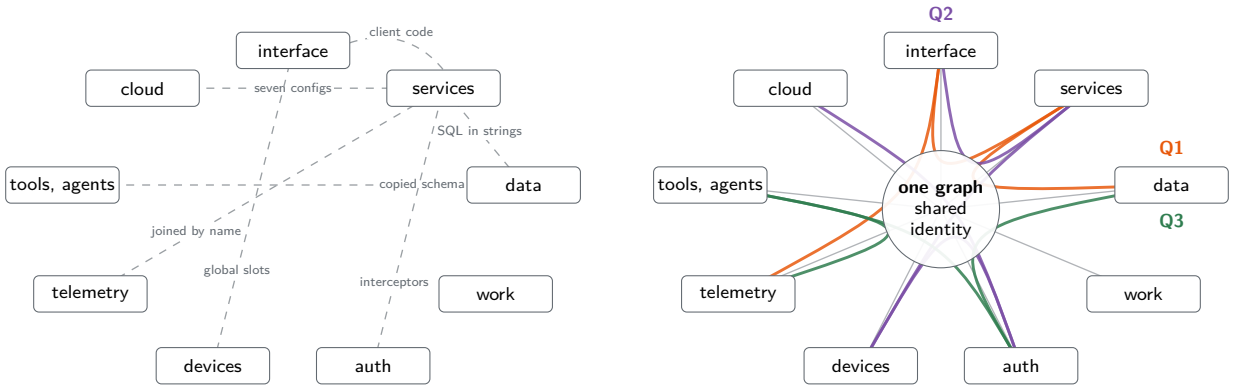
Interface. The renderers draw route and container nodes with Compose on Android, iOS and the desktop, and with React on the web. `surface` expresses menus, tabs, tables, toggles and other widgets as reducer state machines that depend on no toolkit, and `surface-compose` renders them. On the graph a widget's behaviour is a small TRANSITION net inside a Surface, a statechart in the sense of Section 4; rendering is the fold of Section 8.3; and each renderer is a reconciler of render plans. `ui` lets authors give interface elements identities so that tools can find them, and on the graph those identities are stable element identifiers. The embedded web runtime already hosts pages in a session with local bundles, a content policy and a typed bridge. On the graph the web view is a place nested in the app, its bundle is an artefact with a digest, its bridge is a trunk whose host imports are capability ports, and its content policy is a set of grants. Finally, `blueprint` already draws a running application's graph as cards with typed pins, connected by wires and grouped in frames, on the canvas that `compose-flow` provides. It is read-only today. With RG the same canvas becomes an editor for graph-owned scopes, because an edit is a patch that the kernel checks, which is the Blueprint editor of Section 4 extended to the whole system.

Services, data and work. `service` describes operations with typed requests and responses, chains of interceptors and operation descriptors whose port type the runtime already uses. On the graph each operation is a SESSION with a contract fingerprint, each interceptor is an interceptor incidence, and the HTTP and WebSocket transports of `io` are realizations. `db` covers object stores with cache and retention policies, a SQL language, SQLite in the browser, a GraphQL client and a graph-database client whose Cypher queries are checked for tenant safety. On the graph these are stores and datasets, retention is an attribute of a dataset, the tenant check is rule A6 compiled into each query, and traversals over data compile into SQL or Cypher (Section 8.3). The data graph that the current runtime approximates with heuristics is declared: definitions declare stores and datasets, and the compiler derives each operation's ACCESS footprint from its typed queries. Existing databases are imported by introspection adapters that record provenance, so a schema found in Postgres is *observed* until a definition claims it. A schema change is a patch to datasets and REFERENCES, and the matching data migration is induced by a functor between schemas [314], executed at the owning store. `work`, which is being rebuilt around intents, claims, checkpoints and receipts, is the durable profile: its steps are durable transitions, its receipts are observations, and recovery replays committed outcomes. Actors are exclusive components with mailbox channels. `notification` and `google` provide channels to devices and to Pub/Sub topics, and connectors to external services; a connector's consent is a grant.

Security. `auth` is already graph-native. On the server it is a graph of about fourteen nodes for tokens, sessions, personal access tokens, service accounts and delegated exchange, and its secured ports check an authorization requirement before a call proceeds. On the graph its principals, memberships and tenants are Principals and attributes, a secured port is one whose SESSION requires a GRANT, and the authority traversal finds every effect reachable without one. `security` encrypts conversations end to end with MLS, implemented in C++ behind a C interface that Kotlin/Native, JNI, WebAssembly and JSI reach. On the graph it is a native component behind trunks, and the boundary of plaintext is a boundary of labels: only components on members' devices read plaintext, and moderation sees only what a member reports (Section 17.1).

Modules	Status	Elements of Σ_{std}	Traversals and executors
Kernel and language			
`core`: process-wide slots, capabilities, fact provenance	implemented	slots become capability ports offered by places; fact origins become evidence origins	support resolution (Section 6.6)
`graph-port`, `graph-runtime`: ports, definition IR, wiring, navigation, actors, shapes	implemented	the kernel's tables; Bind, Session, Transition; Route; exclusive Component	injection, navigation and lifecycle traversals
`compiler`, `flexbuffer`: generated codecs, FlexBuffers	partial	Contracts and fingerprints; `rgp/1` frames	codecs; compilation plans (Section 10.7)
`ffi`: trunks to C++ and Hermes	prototype	a Place per runtime; native Components; ownership modes	trunk plans; fingerprint gate
`dependeasy`: build plugins	implemented	Artifacts related to sources by Derives	the build graph as a projection
Interface			
`graph` renderers, `ui`, `surface`, `surface-compose`	partial	Surface, Route; signal Channels; behaviours as Transition nets	render fold; reconciler; route table
`web`: embedded web runtime	partial	a nested Place with capability ports; bundles as Artifacts	bridge plans; content policy as Grants
`blueprint`, `compose-flow`: graph canvases	partial	—	views (Proposition 2); edits as patches
Services, data and work			
`service`, `io`: operations, routes, transports	implemented	Session contracts; Routes; realizations over HTTP and WebSockets	binding plans; contract compatibility
`db`: object, SQL, GraphQL and graph stores	partial	Store, Dataset; Access, References, Derives	query plans; migrations as $\Delta/\Sigma/\Pi$ [314]; traversals compiled into stores
`work`: durable work	partial	durable Components and Transitions; receipts as Observations; schedules	run plans; recovery from committed outcomes
Actors, in `graph-runtime`	partial	exclusive Component; mailbox Channel; keyed Realizes; partitioned Places	virtual activation; fencing
`notification`, `google`: push, Pub/Sub, connectors	partial	Channels to devices and topics; external Components	delivery plans; consent as Grants
Bus and Mesh	design	Resource; topic and queue Channels; multiparty Session	lints: dead letters, idempotency, ordering keys
Security			
`auth`, `auth-core`: principals, tenants, tokens, secured ports	implemented	Principal; Grant, Access; tenancy (A6)	authority and flow analysis; reference monitor
`security`, `crypto`, `secrets`: end-to-end encryption, primitives, secret stores	partial	labels and declassifiers; native Components; Resources with a secret label	label flow (W11); rotation plans
Device capabilities			
`media`, `location`, `sensors`, `health`, `tactile`	partial	capability Contracts offered by device Places; personal labels	support resolution with fallback; permission requests as plans
Observation, tools and agents			
`telemetry`, `performance`, `devtools`	implemented	Observations; Observes, Derives; budgets as Claims	instrumentation and fault plans; trace joins by identity
Capture and replay	design	check Component, Claim; Attests; facets marking where a capture cuts a connection	capture regions; replay plans
`tooling`, `code`, `idb`, `cli`	implemented	tools as Sessions with effect classes	queries, plans and receipts behind every face
`mcp`: tool protocol	partial	tools that read by default; effects under grants	the change pipeline (Figure 15)
`conductor`: agents working together	implemented	agent Principals; workflows as durable Transition nets with gates; evidence as Claims	proposal, check, plan, receipt
Workbench	partial	—	each pane is a view (Proposition 2)
Cloud			
`cloud`: control plane	partial	Place, Resource, Artifact; Realizes, Places	reconcilers; plans with approvals
`cloudflare`: Workers runtime	implemented	isolates and Durable Objects as Places; realizations of sessions, channels and stores	the generated deployment layer

Table 20: Reaktor's modules and tools as fragments of Σ_{std}. Status describes each module as it is today, from its source, tests and documentation; a group marked partial contains at least one partial module. The other columns describe the design, which no module implements yet.



(a) Today: modules are joined pair by pair, by integration code, names and copies, where they are joined at all.

(b) With RG: every module joins through shared identity, and a question that spans modules is one traversal. Q1–Q3 follow the first three questions of Table 21; modules not drawn are omitted.

Figure 23: The modules of Reaktor, joined in pairs today and through one graph with RG.

Secrets are resources with a secret label, and `crypto`'s primitives are pure components.

Device capabilities. Camera, gallery, audio, speech, location, step counting, health records and haptics are adapters behind slots today, each with its own check of availability and its own permission request. On the graph each is a capability contract offered by device places, with availability as part of the offer. A feature that needs a capability states the requirement, and support resolution chooses a provider, or a declared fallback, before the feature activates (Section 6.6). A permission request is a step of a plan whose outcome is a grant from the person who owns the device. Health and location data carry personal labels, so their flows are checked by W11 like those of any other personal data.

Observation, tools and agents. `telemetry` already walks the port graph with a visitor and installs tracing interceptors on ports; on the graph that visitor is the telemetry traversal of Table 8, and its result is a patch. `performance` defines budgets and measurement requirements for every target and fails a gate when a required metric was never recorded. On the graph a budget is a claim, a measurement is evidence, and a missing measurement is a gap in coverage. `devtools` streams the graph's shape, port calls, traffic, logs and crashes to the workbench, and injects faults through port interceptors. On the graph its taps are observation plans, its faults are the fault plans of Section 19, and its streams are observations joined by identity. `tooling`, `cli`, `code` and `idb` provide discovery, task plans, runs, approvals and receipts, supervised processes, code intelligence and device automation, and `mcp` exposes tools that read by default and need an authenticated host and an explicit effect to write. On the graph each tool is a SESSION with an effect class, under grants, and agents read the graph through the same typed queries as tools and people, exposed as tools of the Model Context Protocol [18]. `conductor` runs coding agents from several vendors over one checkpointed thread, composes them in workflows of agent, check and gate stages, and keeps evidence that links tasks, candidates, findings and checks. On the graph its workflows are durable transition nets with human gates, its agents are principals that propose, and its evidence is claims and attestations. Agents act only by proposing patches, which are checked, planned and admitted like any other change, under a GRANT that never includes approving one's own plan (Figure 15). What an agent knows about a system is then what the graph and its coverage claims say, not what it scraped from files.

Cloud. `cloud` models operations, runs, events and approvals across Pulumi, Dagger and Wrangler. `cloudflare` runs Kotlin/JS on Workers with routing, D1, R2, KV, Durable Objects, service bindings and AI bindings, has clients for Postgres, ClickHouse and Memgraph, and mounts the graph's service nodes into its router. On the graph, places and resources are elements, the configuration of each provider is generated from the deployment layer (Listing 19), and the operations of the control plane are plans with approvals, executed and reconciled like any other.

14.2 When everything is connected

A module on its own answers questions about itself. Connected through one graph, the modules answer questions that span them, and those are the questions builders actually ask. With about forty modules there are 780 pairs, and every pair that shares identities can answer questions that neither answers alone. Today each such pair is joined by integration code, if it is joined at all: the survey found up to seven models of one system inside one tool, joined by names, and four observation paths whose identities do not match (Table 28). In RG the join needs no pairwise integration code, because identity is shared by construction. Its cost moves elsewhere: to declaring elements, footprints, labels and grants, and to carrying identity through every runtime. Figure 23 contrasts the two arrangements, and Table 21 lists ten questions that span modules, each answered by traversals of the connected graph.

The builder's power. Connecting the modules changes three things for the person or agent building on Reaktor.

Question	Modules it spans	Traversal
Which screens, routes and dashboards break if this column is dropped?	data, services, interface, telemetry	reverse reachability over Access, Session and Derives
Can this feature be pushed to phones today, and to which ones?	interface, services, device capabilities, auth, cloud, FFI, assurance	closure, support resolution, admission
Which personal data can reach the LLM provider, and through which components?	data, auth, agents, tools, telemetry	label flow (W11) over Derives, Session and Channel
Why did checkout take three seconds for this user?	telemetry, devtools, services, data, cloud	critical path (max-plus) over observations joined by identity; provenance
What does this agent's proposal touch, and who may approve it?	agents, auth, work, and every module the patch touches	impact; authority; A5
Which notification goes out when an order is refunded, to which device, under whose consent?	work, notification, auth, device capabilities	paths over Transition, Channel and Grant
What breaks if this secret is rotated, and in which order must its users change?	secrets, auth, cloud, services	reverse reachability; an ordered plan (fold)
Which tests must run for this patch, and on which devices?	tools, device capabilities, services, interface	impact; support resolution to select devices
What degrades if the analytical store is down?	data, services, telemetry, tools	reverse reachability over Derives and Access, by failure domain
Does this feature work on this device, and what does it fall back to?	device capabilities, interface, FFI	support resolution with fallback

Table 21: Questions that span modules, each answered by traversals of the connected graph.

- *Questions become queries.* Each question in Table 21 is a traversal with a coverage claim. Today each is an investigation across repositories, dashboards and people.
- *Changes become one patch.* Renaming a field touches a dataset, a contract, a screen, a dashboard, an analytical table and a graph projection. On the graph it is one patch, reviewed once as a graph diff, with its impact computed and its migration generated, rather than six coordinated changes in six tools.
- *New modules join without adapters.* A new module adds derived kinds and traversals, and once it declares its elements and footprints it takes part in every existing query. A payments module, for example, receives authority analysis, label flow, telemetry, code push, deletion closure and test selection without writing an adapter for any of them.

The value of each module therefore grows with the number of modules it connects to, as in a network. This is also why the graph matters more for agents than for people: an agent can answer a question that spans forty modules with one query, instead of reading the code of forty modules.

14.3 Truthful state

R17 asks for code and its ecosystem as a database with transactional edits. That helps only if code, processes and the graph never disagree, and "never disagree" needs a definition that can be checked.

Definition 10 (Truthful system)**.** A system is *truthful* at a revision vector v, which gives one revision per consistency domain, if it satisfies four conditions:

(i) every source-owned scope's definition region has the digest of the elaboration of its source at v;
(ii) every observed activation has a Realizes assertion to an element of an admitted revision, under an epoch issued by its authority;
(iii) every managed resource realizes a definition element, and every unmanaged one is imported with provenance and marked observed, never desired;
(iv) every claim's status is a function of its evidence and of the dependency closure of that evidence at v.

Condition (i) is checked in CI by digest equality. Condition (ii) holds by construction on hosts that admit only releases whose digests are in Γ; anywhere else, an activation without a realization is reported as of unknown provenance, not inferred. Conditions (iii) and (iv) are reconciler and invalidation traversals. Heuristic recovery of structure, such as regular expressions over SQL or matching spans to nodes by label, becomes unnecessary for managed parts and explicitly labelled for imported ones. Human and agent changes take the same path, each step an element of the graph. A snapshot and query yield a proposal (a patch artefact). The proposal is checked and planned, and the plan gets a fingerprint. A Grant approves it, and no principal may approve its own proposal. An executor applies the plan. A reconciler compares the result, and the receipt is recorded as an Observation.

15 Assurance inside the graph

R16 asks for guarantees that are part of the system rather than documents beside it. In RG a Claim is an element. It has a subject, a proposition, assumptions and bounds. Attests relates it to evidence (observations and artefacts, each with a method and a coverage) and to its verifier. Status is a vector, not a colour. The dimensions are origin (desired,

```
val refundsOnce = claim("refunds.at-most-once", subject = refunds) {
    proposition(AtMostOnce(effectOf(refunds)))
    assume(TimeoutMeansUnknown, DeduplicatedAt(provider(refunds)))
    bounds(maxRetries = 2, maxDuplicates = 1)
    verifier(Tlc(template = SessionCall))
    requiredBy(release(region("support")))
}
```

Listing 20: A claim about a SESSION call (proposed API).

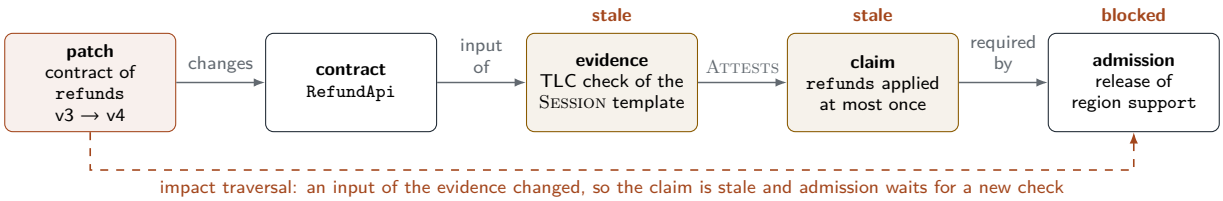


Figure 24: Evidence is a function of its inputs. A patch to one input makes the evidence stale; the impact traversal carries this to the claim and to every admission that requires the claim (Section 15).

source-derived, recorded, observed, simulated, checked, attested), freshness, coverage and conflict. Evidence kinds are deliberately *not* totally ordered. A production trace cannot discharge a universal "no unauthorised write" claim, and a proof about a model cannot discharge a latency objective [295].

Listing 20 declares a claim about the refund call of Listing 23.

Invalidation. Evidence is a function of its inputs: artefacts, revisions, workload, fault schedule and assumptions. When a patch changes an element in the dependency closure of a piece of evidence, the impact traversal (Table 8) marks the claim stale. This is a build system's rule for rebuilding a target whose inputs changed [251], applied to assurance. Figure 24 traces one invalidation, and Listing 21 shows the claim's status afterwards.

Specifications are generated. Four tools cover four kinds of claim.

- **TLA$^+$.** The specification of a protocol region is assembled from family templates by (1). Model checking then runs on the same identities that the runtime reports. Five templates exist as hand-written TLA$^+$ modules, checked by TLC (Table 34): effect identity for SESSION calls, ownership handover for exclusive components, region release, the outbox behind a CHANNEL, and the watch of RGP (Section 12.4). Generating them from the graph is planned work.
- **Alloy.** The static judgments of Section 6.5 are first-order constraints over the C-set schema, which is already an Alloy signature, and are checked for consistency and for counterexamples [178].
- **Lean 4.** The kernel's metatheory is to be mechanised once [85]: preservation of well-formedness by each patch operation (Proposition 9), determinism of traversals (Theorem 6) and functoriality of the semantics (Theorem 3).
- **Testing.** Implementation conformance is tested by deterministic simulation in the style of FoundationDB and P [92, 366], and by replaying captured boundaries.

The precedents for each are industrial: TLA$^+$ at AWS [264], verified distributed systems such as IronFleet [165], and simulation testing at FoundationDB.

Claims gate change. Release admission requires fresh evidence for the candidate's digest. Approval binds the plan's fingerprint, and any drift invalidates it. The policy that decides which claims a release needs is outside the authority of the patch being admitted, so a program cannot weaken its own gate and certify itself. The bounded models also feed the language. Because the effect-identity model shows that reading a timeout as "not applied" is unsound, RGL's Outcome has an explicit UNKNOWN. The handover model shows that a check by the sender, even one made just before the write, does not stop a delayed write, while a check by the store does (Table 34). Therefore `atomic(store)` checks the writer's epoch inside the store's transaction.

```
claim refunds.at-most-once          subject  Session support.refunds
  origin     checked by TLC from the SessionCall template
  freshness  stale: patch p-1842 changed contract RefundApi v3 -> v4
  coverage   bounded: retries <= 2, duplicates <= 1
  conflict   none
  gates      release of region support, blocked until re-checked
```

Listing 21: The status of the claim of Listing 20 after the patch of Figure 24, as a reviewer sees it.

	Problem	outcomes with UNKNOWN	effect identity	epoch fencing	outbox in `atomic(store)`	QoS and qualified bindings	traversals over the system	contracts and fingerprints	patches, closure, admission	lineage and labels	grants and tenancy	observations and claims	generated deployment
P1	Unknown outcomes	●	○									○	
P2	Duplicate effects	○	●			○							
P3	Retry amplification, metastability					○	●					○	
P4	Deadlines that do not compose	○				○	●						
P5	Tail latency and capacity					○	●					○	
P6	Gray and partial failure						○					●	
P7	Partitions and split brain			●		○						○	
P8	Dual writes		○		●								
P9	Stale owners			●					○			○	
P10	Derived data and deletion						○			●			
P11	Schema evolution							●	●				
P12	Ordering assumptions					●							
P13	Version skew during upgrades							●	○				
P14	Configuration errors					○			○				●
P15	Topology drift											○	●
P16	Code pushed to clients			○				○	●				
P17	Interaction between systems					○		●		○			
P18	Emergent misbehaviour						●					○	
P19	Ambient authority						○			○	●		
P20	Diagnosis across runtimes								○			●	

Table 22: Twenty problems of multi-runtime systems and the RG mechanisms that address them (● primary, ○ contributing).

16 Problems of multi-runtime systems

The fallacies of distributed computing were listed in the 1990s: that the network is reliable, that latency is zero, that bandwidth is infinite, that the network is secure, that topology does not change, that there is one administrator, that transport costs nothing, and that the network is homogeneous [303]. A multi-runtime system meets all eight and adds its own. Each runtime has its own failure model, deployment unit, scheduler and debugger, and each runtime's guarantees stop at its edge.

Studies of production failures show where such systems break in practice.

- In 198 failures of five data-intensive systems, 92% of the catastrophic ones came from incorrect handling of errors that had been explicitly signalled, and almost all could be reproduced on three nodes or fewer [362].
- Across 597 outages of 32 cloud services, upgrades, network faults and bugs led the known root causes [158].
- Network partitions caused 136 failures in 25 systems, most of them catastrophic and deterministic, and many triggered by isolating a single node [13].
- About two thirds of 123 upgrade failures came from data that two versions interpreted differently [365].
- Configuration is a dominant cause of failures [357]; at Facebook it caused 16% of high-impact incidents [326].
- A fifth of the studied incidents of three large clouds came from interactions between systems [327].

This section selects twenty problems that recur in these studies and in the survey of Section 20.2. For each problem it states the *basic implementation*, which is what code does without a model of the system: each call site, service and runtime handles the problem locally, by convention. It then states how the basic implementation fails, and what the graph changes. Table 22 maps the problems to the twelve mechanisms that address them. Most problems need two or three mechanisms, and most mechanisms serve several problems.

16.1 Failure and time

P1 Unknown outcomes. *Basic:* a call that times out, or whose connection drops, is reported as a failure, and the caller retries or gives up. *Failure:* a timeout cannot distinguish a slow peer from a dead one, so the effect may have happened. Reading the timeout as "not applied" tells the user that a payment failed when it succeeded; TLC finds that trace in eight steps (Table 34). *With RG:* a Session call returns an Outcome with four cases: success, domain rejection, failure and UNKNOWN. W5 rejects an effectful call whose outcome type lacks UNKNOWN, and a reconciler resolves unknown outcomes later by querying the provider for the effect identity.

P2 Duplicate effects. *Basic:* each call site wraps its call in a retry loop. Idempotency keys are optional, are sometimes generated per attempt rather than per effect, and are rarely bound to the request they protect. *Failure:* a retried charge is applied twice, or a reused key silently returns the result of a different request. A unique request identifier in the contract is what makes retries safe [131]. *With RG:* effect identity is a QoS dimension of the Session, and its strongest level binds the key to the digest of the request. A binding qualifies only if the provider deduplicates at that level (A2), and the key is minted once per logical effect, outside the retry loop.

P3 Retry amplification and metastable failure. *Basic:* every layer retries on its own. *Failure:* retries multiply along call paths. Three layers that each make up to four attempts send up to 64 requests to the bottom layer for one request at the top. Under overload this extra work can keep a system down after the trigger has gone, a *metastable* failure [39]; the standard advice is to retry at one layer only, with backoff, jitter and a retry budget [40]. *With RG:* retry policies are attributes of Session bindings, including retries inside a database driver once they are declared, and redelivery is an attribute of a Channel. The worst-case amplification along one path is then a path problem over a max-times semiring, computed at plan time and checked against a budget (Listing 22). The load that reaches a shared resource sums over every path and every fan-out that leads to it, which is the same traversal over the counting semiring of sums and products. A cycle of retrying calls has unbounded amplification and is reported as such.

P4 Deadlines that do not compose. *Basic:* each hop has a fixed timeout, chosen locally. *Failure:* an inner timeout longer than the outer one keeps working after the caller has given up, and inner timeouts whose sum exceeds the outer deadline make retries pointless. *With RG:* a deadline travels with the call. A traversal checks that every callee's deadline fits within its caller's remaining budget, and the max-plus semiring gives the worst-case latency of every acyclic call chain (Section 8.2). Expiry after the request may have been sent yields UNKNOWN, never failure; expiry before it was sent is a known failure.

P5 Tail latency and capacity. *Basic:* a request fans out to many shards and waits for all of them, and capacity is planned from estimates and staging tests. *Failure:* the slowest shard sets the latency. If each server is slow on one request in a hundred, a request that fans out to a hundred servers is slow 63% of the time [87]. Bottlenecks often appear only under production traffic, which is why some operators test capacity by shifting live traffic [345]. *With RG:* fan-out and join policies are explicit in code entries (Listing 8). A hedging policy is an attribute that qualifies only on idempotent reads. Latency and capacity are path problems over observed values, so a placement or a release can be predicted before it ships and compared with observations afterwards.

P6 Gray and partial failure. *Basic:* each process has a health check. *Failure:* a component can look healthy to its failure detector while its clients fail. This *differential observability* is what defines gray failure [174], and partial failures of this kind often need production conditions to appear [229]. *With RG:* observations are joined to incidences, not only to processes, so a connection's health is measured from both of its ends. A disagreement between a component's own health and what its clients observe is a finding, and a claim about liveness can be stated per connection.

P7 Partitions and split brain. *Basic:* partitions are assumed to be rare, and a leader holds a lease. *Failure:* partitions lose data, bring deleted data back and break locks [13], and independent testing keeps finding systems that violate their advertised guarantees under partitions [196]. *With RG:* each Channel and Session declares its levels of delivery, order and effect identity (Table 6). A binding qualifies only if partition tests of its provider found no violation of those levels. During a partition a provider may become unavailable, so that calls end in failure or UNKNOWN, but it must not weaken its levels; a test can refute that, never prove it. Exclusive components are fenced by epoch at the stores they write (Figure 4). Places record failure domains, so partition tests can be generated from the declared guarantees instead of written by hand.

16.2 State and data

P8 Dual writes. *Basic:* a service writes its database and then publishes an event, or publishes first. *Failure:* a crash between the two loses an event or announces a write that never committed, and concurrent writers to two stores race and leave them permanently inconsistent [198]. *With RG:* `atomic(store)` commits the effect and its outbox record in one transaction, and the record is published only after the commit. The outbox model of Table 34 shows the dual write losing an event in one step, while the outbox holds.

P9 Stale owners. *Basic:* a component holds a lease and checks it before each write. *Failure:* a holder that pauses past its lease writes after a new owner has taken over (Figure 4). Checking the lease just before writing does not help; TLC finds the trace in three steps. *With RG:* the store checks the writer's epoch inside its own transaction, so a stale write is rejected where it would take effect. Activations carry their epochs, and observations report them.

P10 Derived data and deletion. *Basic:* caches, search indexes, analytics copies in ClickHouse and graph projections in Memgraph are kept up to date by jobs and scripts. *Failure:* reads are stale for an unknown time, and a deletion misses a copy that nobody remembered. *With RG:* every copy is a dataset related to its source by Derives, with lineage. Invalidation is a traversal, and freshness is a declared attribute. Rows are not part of the graph, so a deletion is a data plan: its targets are the closure of the deleted data over Derives, and it runs as a saga with one receipt per store. Labels follow Derives, so a copy of personal data is personal.

P11 Schema evolution. *Basic:* migrations run separately from code deployments. *Failure:* during a rollout, code and schema disagree. F1 showed that safe online schema change needs intermediate states, such as delete-only and write-only, and servers at most one schema version apart [284]. *With RG:* a schema change is a patch to datasets, and its data migration is induced by a functor between schemas (Section 14). The expand and contract steps are separate revisions, and each intermediate revision must be well formed. Contract compatibility between the code and each schema is computed, not assumed. F1 also bounds version skew with leases, which phones cannot give; for clients, the bound is the oldest version that admission still accepts.

P12 Ordering assumptions. *Basic:* code assumes that a queue preserves order. *Failure:* some queues order only on a best-effort basis (Section 13), so consumers see reordered events in rare cases and corrupt state silently. *With RG:* order is a QoS dimension of the Channel, ordered from none to per key to total. A binding to a queue that does not provide the declared order fails A2 before activation.

16.3 Change

P13 Version skew during upgrades. *Basic:* rolling upgrades, with the hope that versions are compatible. *Failure:* mixed-version operation can last a long time [8], upgrades lead the causes of outages [158], and most upgrade failures come from data that two versions read differently, which pre-release tests usually miss [365]. *With RG:* every boundary carries contract fingerprints. Compatibility is computed for each pair of versions that admission may combine, a release region is closed under the relations that must change together (Equation (3)), and the window during which two versions coexist is explicit in the plan.

P14 Configuration errors. *Basic:* configuration files and environment variables, kept per runtime and copied between them. *Failure:* misconfiguration is a leading cause of failures [357], which is why Facebook manages configuration as code with validation, review and canaries [326]. *With RG:* configuration is typed attributes of elements. The deployment layer is generated from one graph (Listing 19), and a configuration change is a patch that is checked, reviewed as a graph diff and released in stages like code.

P15 Topology drift. *Basic:* what runs where is recorded in deployment scripts, dashboards and documents. *Failure:* the records disagree; one surveyed product declares its topology in seven hand-kept places (Section 14). *With RG:* topology is the deployment layer of the graph. Reconcilers compare it with what is observed, and the truthful-state conditions of Definition 10 make agreement checkable.

P16 Code pushed to clients. *Basic:* app store releases, plus a code-push service that ships a JavaScript bundle. *Failure:* a pushed bundle calls a host interface that the installed app does not have, or arrives in the middle of a session, and there is no principled rollback. *With RG:* a push is a region release. Its closure includes everything that must change together, admission checks fingerprints at the trunk on each host, and next-entry or drain-and-swap activation is fenced by epoch (Section 9.5).

16.4 Interaction, authority and diagnosis

P17 Interaction between systems. *Basic:* each system is tested alone, against mocks of the others. *Failure:* interactions between systems caused a fifth of the cloud incidents studied in [327], and many of the cross-system failures it studied in open-source systems involved the data that passed between them. *With RG:* the boundary between two systems is a typed hyperedge with contracts on both sides. A third-party system is imported as an observed scope with provenance, so the boundary is checked like any other, and labels and lineage cross it.

P18 Emergent misbehaviour. *Basic:* independent control loops, such as autoscalers, retry policies, caches, circuit breakers and rebalancers, are each tuned on their own. *Failure:* together they synchronise, oscillate or livelock without any component being at fault [249]. *With RG:* controllers and reconcilers are components whose reads and writes are in the graph. A traversal finds cycles among control loops that read and write the same elements and reports each cycle with its members, so that a person can assign it a damping policy or a single owner.

P19 Ambient authority. *Basic:* a service acts with its own authority on behalf of whoever calls it. Secrets live in environment variables, and the tenant is a parameter passed by convention. *Failure:* a caller makes a privileged service act on a resource it could not reach itself, the confused deputy [162]; secrets leak through logs; and one missing filter exposes another tenant's data. *With RG:* authority is a Grant, and the authority traversal finds every effect reachable without one. Tenancy is rule A6, and secrets are resources with a secret label that W11 keeps out of logs and telemetry. An agent is a principal like any other, under the same rules (Section 17.1).

P20 Diagnosis across runtimes. *Basic:* trace identifiers are propagated by instrumented libraries, as in Dapper [311], and spans are attributed to code by service and operation names. *Failure:* a runtime without the library breaks the trace, names attribute spans to the wrong code, and failures that would reproduce on three nodes are never reproduced [362]. *With RG:* instrumentation plans are generated for every connection, observations are joined by element identity and revision, and a trace with gaps states its coverage. The question "what changed?" is a query over patches, and capture and replay at boundaries turns a production failure into a test.

```
val amplification = traversal<Double>("sessions.retry-amplification") {
    select(Session, from = Client, to = Server)
    semiring(MaxTimes)
    along { call -> call.binding.retry.maxAttempts.toDouble() }
    finding(where = { path, factor -> factor > path.entry.budget.amplification }) { path, factor ->
        "amplification $factor on ${path.render()} exceeds ${path.entry.budget.amplification}"
    }
}
```

Listing 22: Worst-case retry amplification along one path, as a traversal over a max-times semiring (proposed API).

16.5 One incident, with and without the graph

Consider a hypothetical incident in the platform of Figure 2. A slow query plan raises the latency of feed reads from Postgres. The phone retries a feed request up to three times; the feed Worker retries its Postgres session up to three times; and the database client inside it retries once. A single tap can therefore become $4 \times 4 \times 2 = 32$ queries. The connection pool saturates, and latency rises for every route that shares it. When the slow plan is fixed, the retries keep the load above capacity, so the failure is metastable [39]. Meanwhile the database's health check stays green while the edge sees timeouts, a gray failure [174]. Engineers join logs by time and guess.

With the graph, three findings exist before the first deployment, provided the driver's retries are declared as an attribute of the database session's binding. The amplification traversal of Listing 22 reports a factor of 32 on the path from the phone through the feed Worker to Postgres, against a budget of 4, and names the three bindings. Its counting variant sums the load over every route that shares the pool. The deadline check reports that the database session's deadline exceeds the feed's own. The shared pool appears as one Resource that several routes depend on, so its saturation is a single point of contention that the impact traversal can show. In production, observations joined to the session incidence show that the client side and the server side disagree. The gray failure is then a finding with a witness, not a guess.

16.6 What the graph does not fix

The graph changes where problems are found, not the physics that causes them. Networks still partition, disks still fail and providers still have correlated outages. Bugs inside component bodies stay bugs. Declarations can be wrong: a QoS level or an idempotency claim that a provider does not meet is caught only if qualification tests it. Approvers can still approve a bad plan, and every check has a cost that Section 20.4 must show to be worth paying. What the graph does provide is a place where the twenty problems above are stated once, checked before activation where they can be, and observed by identity where they cannot.

17 Systems in scope

Multi-runtime systems differ along eight axes:

- who owns state, and how it is partitioned;
- the consistency each datum needs;
- the latency regime;
- what an effect is, and when it becomes final;
- how much people and agents take part;
- how tightly the system is coupled to the physical world;
- how many tenants share it;
- how its computation is produced and trusted: written, simulated or learned, and how its results are reproduced.

Table 23 selects eleven classes that together span these axes. For each it gives the difficulty that defines the class, the tools that address part of it today, and the RG mechanisms that address it as a whole. Section 17.1 then takes the classes one at a time.

What the classes share. Every class needs the same five things across its whole system:

- one model of what exists;
- identity that survives restarts and runtimes;
- effects that are neither lost nor misreported;
- actions attributed to principals under grants, with data flowing only where its labels allow;
- placement chosen from capabilities.

Today each is solved inside one tool: order inside a log, durability inside a workflow engine, isolation inside an actor runtime, permissions inside an API gateway. Each is lost at the boundaries between tools. RG states them once, as types, rules and traversals over the whole system, and keeps the specialized tools as realizations. This is the sense in which RG is designed to serve these classes better than any single existing system. It is not deeper along any one axis; each cited tool is. It aims to be consistent across the axes at once, where present systems are joined by convention. Whether

Class (examples)	Defining difficulty	Strongest tools today	RG mechanisms
Real-time communication and collaboration (messaging, presence, calls, shared documents)	Per-conversation order and fan-out under churn; offline replay; moderation versus end-to-end encryption	Socket servers, push services, CRDT libraries [202, 310]	Keyed exclusive rooms; signal and mailbox channels with sequence numbers; Replicates with conflict laws; guards that order effects; labels that keep plaintext from moderation sinks
Social and content platforms (feeds, follow graphs, ranking, notifications)	Graph-shaped reads; projections across stores; privacy that must follow derived data; deletion everywhere	Graph databases, caches, search indexes, pipelines	Derives with lineage; labels propagated to derived data; deletion as invalidation across stores; traversals compiled into stores
Transactional commerce and finance (orders, payments, ledgers, inventory)	Correctness under partial failure and external providers; invariants; reconciliation; separation of duties	Workflow engines [329], sagas [140], idempotency keys [219]	`atomic(store)` with outbox; outcomes with UNKNOWN; dataset invariants at commit; transition nets with compensation; grants without self-approval; receipts bound to payloads
Agentic and AI-native systems (tool-using agents, multi-agent work, human-in-the-loop operations)	Non-deterministic, long-running steps; authority for tool calls; prompt injection and exfiltration; evidence of what was done	Agent loops [361], tool protocols [18], control–data separation [74, 89]	Agents as principals that propose; tools as Sessions with effect classes and grants; label flow as a structural defence against injection; durable runs; receipts and claims; context as traversals with coverage
Workflow and process automation (back office, compliance, B2B integration)	Long waits on people and external systems; flaky integrations; audit evidence; deadlines	Workflow engines, robotic process automation, cron	Durable transitions with human sessions as gates; schedules; Attests with digests; completion only with receipts
Interactive simulation and multiplayer games (matches, persistent worlds, live operations)	Latency; authoritative state and interest management; matchmaking; updates without downtime	Dedicated game servers with latency compensation [31]	Keyed rooms placed near players; signal channels with declared QoS; match lifecycles as transition nets; region release with drain-and-swap at match boundaries
Robotics, IoT and fleets (robot fleets, connected devices, vehicles)	Heterogeneous capabilities; intermittent links; safe control handover; over-the-air updates; safety evidence	ROS 2 on the robot [231]; fleet and update services	Devices as places with capability offers; task allocation as a solver; handover with epoch fencing (Table 34); commands with effect identity; safety claims that go stale with firmware; coordination-free traversals
Data and ML pipelines (streaming ETL, materialized views, features, lineage)	Incremental computation; schema evolution; backfills; reproducibility; retention	Logs and stream processors [51, 208]; incremental engines [46, 236]	stream components on Channels, maintained over Z-sets; Derives for lineage; migrations as functors; labels; retention as a dataset attribute
Multi-tenant platforms and control planes (SaaS, bring-your-own-cloud, developer platforms)	Tenant isolation; placement across customer clouds; fleet upgrades; quotas	Kubernetes operators [47], intercloud brokers [360]	Tenancy rule (A6); capability-qualified placement; releases admitted per cell; quotas as grant attributes; observations per tenant and element
Scientific simulation (brain simulators, multiscale models, digital twins)	Models composed across scales and simulators; GPUs and MPI ranks; runs of days that must checkpoint; parameter sweeps; provenance and validation of every result	Simulators [142, 171, 321], model languages [81, 84, 146], co-simulation [97], workflow managers [252]	Populations and projections as elements and hyperedges; simulators as places with capability offers; co-simulation channels with declared delays; durable runs; sweeps as scope factories; provenance as Derives; validation as claims; labels on human recordings
Neurosymbolic AI (learned proposers with symbolic checkers, differentiable logic, program search)	Two representations and runtimes, tensors and terms, joined by glue; trust in learned steps; training signal from checks	Neurosymbolic languages [224, 234], search with evaluators [301, 340]	One typed graph for both halves; message passing that is either a traversal or a neural layer; semiring traversals that compute proofs, probabilities or gradients; proposals checked by rules, solvers and TLC; checks as labelled data

Table 23: Eleven classes of multi-runtime systems and how RG addresses each.

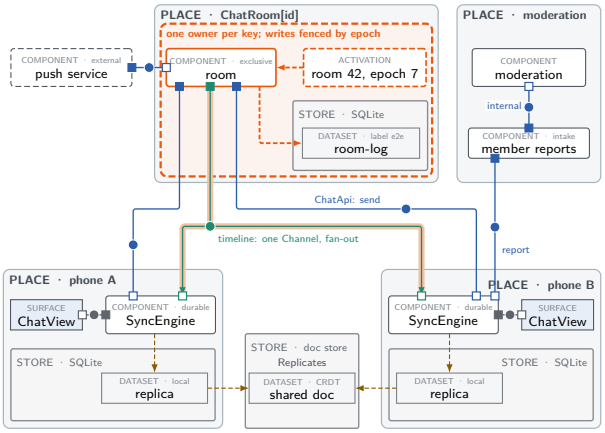
it succeeds is what the hypotheses of Section 20.4 test.

17.1 The classes one by one

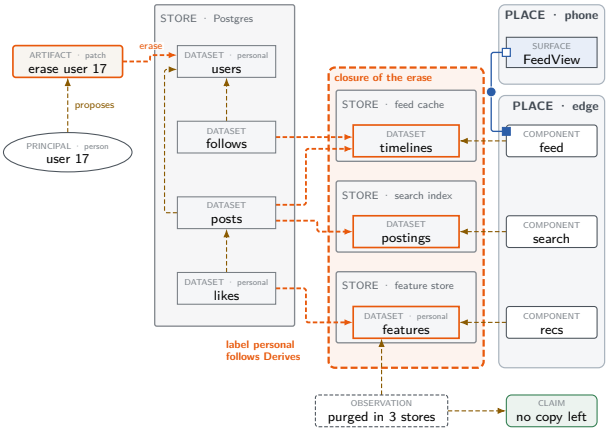
Figures 25 to 27 draw a representative graph for each class, in the notation of Figure 2. Each graph is small, but each shows the structure that makes its class hard and the mechanism that addresses it, in orange. Table 24 names, for each class, one defect that the rules of Section 6.5 catch before activation and that today's tools let through.

Real-time communication and collaboration. This class covers messaging, presence, voice and video calls, and shared documents and whiteboards. It is the running example of Figures 1 and 2, and Figure 25a reduces it to its core. Such systems hold millions of small conversations, most of them idle at any moment and some of them suddenly very busy.

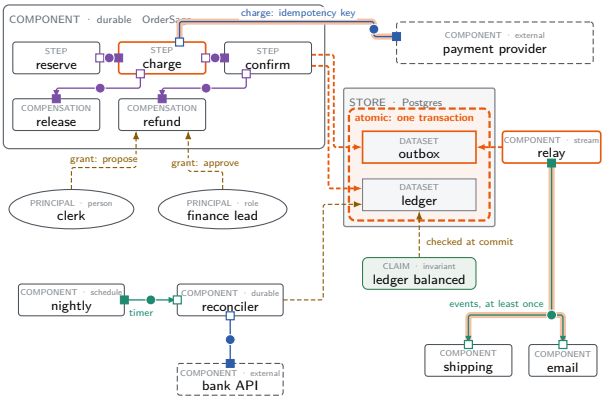
- *What makes it hard.* Messages of one conversation must keep their order while devices connect and disconnect and servers fail over. A device that has been away for a week must catch up without gaps or duplicates. Fan-out reaches phones, browsers and push services, while typing indicators and read receipts may be lost without harm. End-to-end encryption conflicts with moderation and server-side search. Shared documents accept concurrent edits that must converge.
- *Today.* Socket servers, push services and CRDT libraries [202, 310] each solve one of these. Order is the socket server's private business, replay is code in each client, encryption is a library, and moderation is a separate service that has to be told what it may see.
- *With RG.* Each room is an exclusive component keyed by room, realized by a Durable Object, and its epoch fences



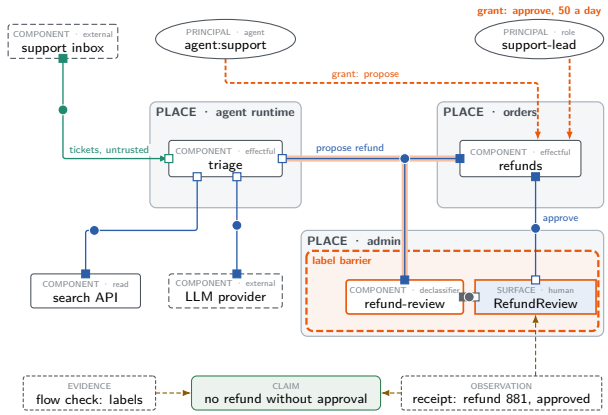
(a) Real-time communication and collaboration. The keyed room owns the order of one conversation, and its activation fences writes by epoch; one timeline CHANNEL fans out to both phones. Moderation never reads end-to-end content on the server: it receives only what a member reports from a phone.



(b) Social and content platforms. Feeds, search and recommendation features are projections, each a DERIVES from the base datasets. Erasing a user is a data plan whose targets are the closure over DERIVES, with a receipt from every store, and the label *personal* follows the same edges.

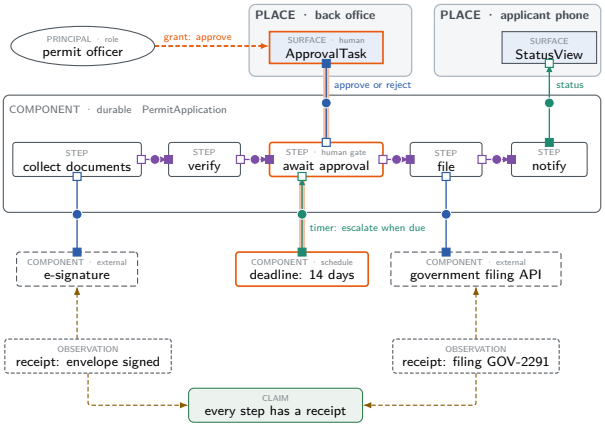


(c) Transactional commerce and finance. The saga's charge carries an idempotency key, and confirm writes the ledger and the outbox in one transaction, so a relay can publish at least once without a dual write. Grants separate who proposes a refund from who approves it.

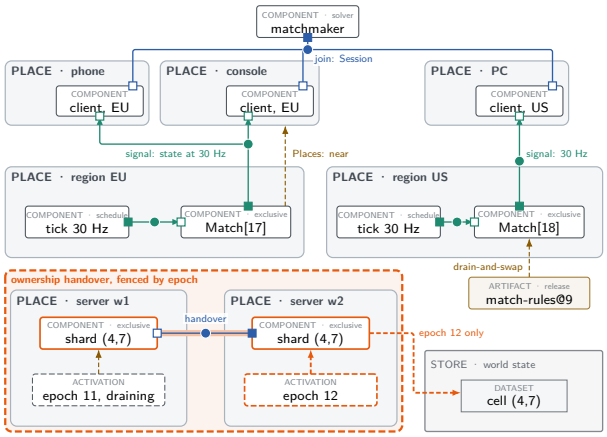


(d) Agentic and AI-native systems. Tickets arrive untrusted; the agent's tools are SESSIONS. The agent holds a grant to propose a refund, not to approve one, and the ticket's text reaches the refund only through the review declassifier, an interceptor on the proposing SESSION.

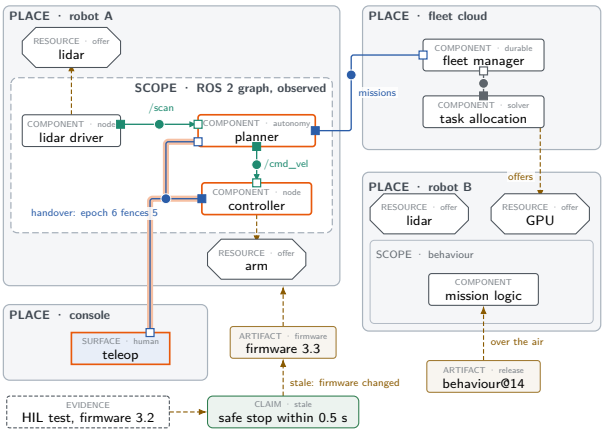
Figure 25: Representative graphs of the classes of Section 17 (1 of 3). Notation as in Figure 2: places and stores are containers, connections end in ports and meet at hubs, dashed arrows are assertions, and orange marks the mechanism that defines the class.



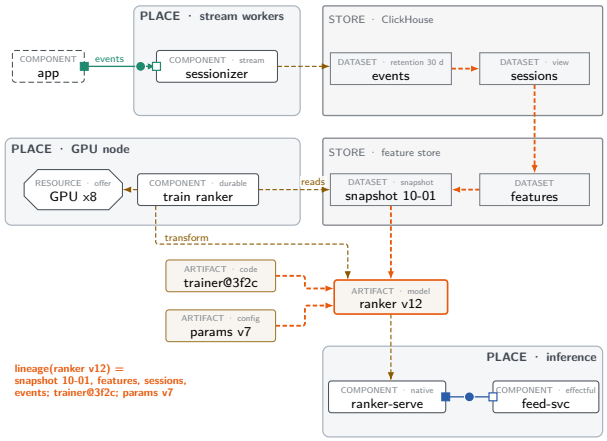
(e) Workflow and process automation. A durable workflow waits on a person through a SESSION to a human surface, a gate like any other call, and on a deadline timer fed by a schedule. External systems are SESSIONS, and their receipts attest the compliance claim.



(f) Interactive simulation and multiplayer games. A solver assigns players to keyed match rooms placed near them; each room ticks and sends state on a signal CHANNEL at a declared rate. A world shard moves between servers by an ownership handover that the store fences by epoch, and new match rules ship by drain-and-swap at a match boundary.

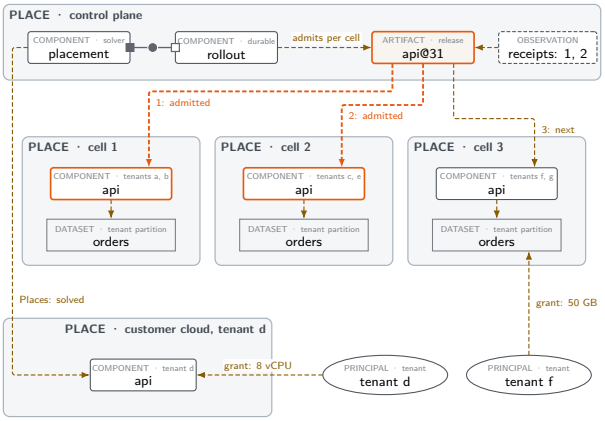


(g) Robotics, IoT and fleets. Robots are places that offer capabilities; the on-robot ROS 2 graph is an observed scope. Control passes between teleoperator and autonomy by an ownership handover in which the controller fences the old epoch. Behaviour ships over the air, and the safety claim goes stale with new firmware.

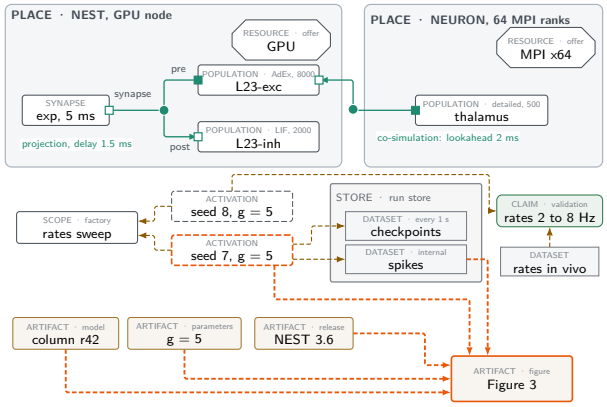


(h) Data and ML pipelines. A stream processor maintains a view, features and a snapshot derive from it, and the model derives from the snapshot, the training code and its parameters. The orange DERIVES chain is what a lineage query returns; retention is an attribute of the raw events.

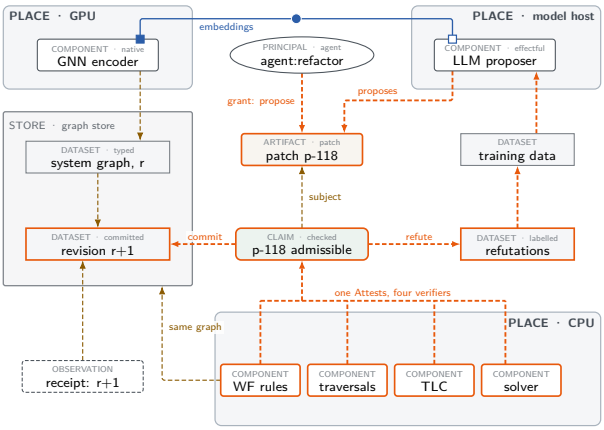
Figure 26: Representative graphs of the classes of Section 17 (2 of 3). Notation as in Figures 2 and 25.



(i) Multi-tenant platforms and control planes. Cells are places, and every workload and dataset carries a tenant facet. A release is admitted cell by cell, each admission a REALIZES of its own; quotas are attributes of grants, and a solver places one tenant's workload in the customer's own cloud.



(j) Scientific simulation. Populations are elements; a projection is a hyperedge with roles pre, post and synapse. The projection between the two simulators is a co-simulation CHANNEL whose minimum delay is the lookahead. The figure DERIVES from the model revision, parameters, seed, simulator release and spikes, and goes stale when any of them changes.



(k) Neurosymbolic AI. A GNN encodes the typed system graph and an LLM, acting for an agent principal, proposes a patch. Rules, traversals, TLC and a solver read the same graph and attest a claim about the patch: if it holds, the patch commits as revision $r + 1$ with a receipt; if not, the refutation becomes a labelled example that DERIVES the proposer's training data.

Figure 27: Representative graphs of the classes of Section 17 (3 of 3). Notation as in Figures 2 and 25.

Class	A defect caught before activation	Rule
Communication	A room's state written by a realization that does not fence by epoch	A2: the binding is not qualified
Social platforms	A deletion that misses a derived store	the deletion patch's closure over DERIVES
Commerce and finance	A charge sent without an idempotency key bound to its request	A2: the binding misses the call's QoS
Agentic systems	Ticket text from a customer reaching a refund's arguments	W11: flow without a declassifier
Workflow automation	A human approval step with no deadline or escalation	the human profile's laws (Table 4)
Simulation and games	Match state shared with a component in another place	A1: a BIND that leaves its process
Robotics and fleets	A motion command whose timeout is read as "not done"	W5: the outcome type lacks UNKNOWN
Data and ML pipelines	A feature table that drops the personal label of its source	W11 on a DERIVES path
Multi-tenant platforms	A query that does not carry the tenant	A6
Scientific simulation	A projection between two simulators with no declared delay	W8: a cycle of value dependencies needs a delay
Neurosymbolic AI	A learned proposer granted the right to approve its own patches	A5: no self-approval

Table 24: One defect per class that the rules of Section 6.5 catch before activation.

writes at the store when it fails over (Table 34). The timeline CHANNEL fans each room's events out to its devices and declares its order and delivery, and a binding that cannot provide them fails A2. A device that has been away does not depend on the channel's memory: it calls the room's SESSION with its last sequence number and receives what it missed from the room's message dataset. Presence and typing indicators travel on a signal channel declared best effort, so nobody builds a guarantee on them. Shared documents declare their conflict laws on REPLICATES. Encryption is a native component behind trunks, and plaintext carries a label that confines it to members' devices. Moderation therefore never reads end-to-end content on a server: it sees only messages that a member reports, released on that member's device under the member's own grant. Server-side moderation, as in the worked example of Section 13.6, applies to rooms without end-to-end encryption. Push to an offline device is a CHANNEL to a component on the device, realized through the platform's push service.

- *Outside RG.* Media codecs, call quality and the convergence algorithms of the CRDTs themselves stay inside opaque components.

Social and content platforms. This class covers feeds, follow graphs, likes and comments, search, recommendations, notifications, moderation and tools for creators. Reads outnumber writes by orders of magnitude, popularity is highly skewed, and most of the data is derived (Figure 25b).

- *What makes it hard.* Many reads are graph-shaped: followers of followers, mutual connections, the audience of a post. The same facts are copied into feed caches, counters, a search index, an analytical store, a graph store and the features of recommendation models, and every copy must be kept current. Privacy must follow the copies: a post restricted to friends must not leak through a cache, an index or a feature, and a block must reach all of them. Deleting an account must reach every copy. Skewed accounts force a choice between writing to every follower's feed and reading at request time.
- *Today.* Graph databases, caches, search indexes and data pipelines each hold part of the picture, and deletion is a script that someone must keep up to date with every new copy (P10 in Section 16).
- *With RG.* Each copy is a dataset related to its source by DERIVES, with lineage. Rows are not part of the graph, so deleting a user's data is a data plan whose targets are the closure over DERIVES; it runs as a saga with a receipt from every store. Labels propagate along DERIVES, so a feature computed from personal data is personal, and its consumers need the corresponding grants. Traversals compile into the stores that hold the data: follow-graph queries into Cypher for Memgraph, engagement analytics into SQL for ClickHouse, and feeds into recursive SQL for Postgres (Listing 3). Writing to feeds or reading at request time is a choice in the plan, made per account from observed fan-out, not a decision buried in code. Moderation decisions are claims with evidence, so an appeal can see what the decision rested on.
- *Outside RG.* The quality of ranking and of recommendation models.

Transactional commerce and finance. This class covers carts, orders, payments, refunds, inventory, ledgers, invoices and payouts. Its systems depend on external providers, such as payment processors, banks, tax and shipping services, that are the sources of truth for money and goods and that can time out after acting (Figure 25c).

- *What makes it hard.* An effect may have happened even when its call failed (P1), and retries can repeat it (P2). Invariants must hold: balances never negative, stock never oversold, every ledger transaction summing to zero. The system must be reconciled against providers that keep their own records. Duties must be separated, so that the person

```
val Support = scope("support") {
    val agent = principal("agent:support", kind = Agent)
    val lead = role("support-lead")
    val tickets = channel<Ticket>("tickets", Mailbox, label = Untrusted)
    val refunds = require(Session.call<RefundApi>(), effect = ReversibleProductionWrite)
    component("triage", ::Triage, profile = Effectful) { consumes(tickets); proposes(refunds) }
    grant(to = agent, on = refunds, actions = setOf(Propose))
    grant(to = lead, on = refunds, actions = setOf(Approve), budget = perDay(50))
    declassifier("refund-review", from = Untrusted, to = Internal, holder = lead)
}
```

Listing 23: An agent's tool under grants and labels (proposed API).

who requests a refund is not the one who approves it, and every step must leave evidence for audit.

- *Today.* Workflow engines [329], sagas [140] and idempotency keys [219] address the calls; nightly scripts reconcile; approvals live in a separate tool; and audit logs are written by convention.
- *With RG.* The checkout of Figure 2 is a saga: a Transition net with compensations. Its charge carries an effect identity bound to the digest of the request, and a timeout yields UNKNOWN, never "not applied". Effects leave through an outbox written in the same transaction (Listing 27). The ledger's balance and the zero sum of each transaction are dataset invariants checked at commit, under serializable isolation or with one writer per account, because snapshot isolation allows write skew to break them. A scheduled reconciler compares the ledger with the bank and resolves outcomes that were left unknown. Separation of duties is a grant rule: no principal approves its own proposal (A5), and a refund above a threshold needs a grant whose budget is an attribute. Receipts are bound to payloads, and TLC checks the call and the outbox templates; their evidence gates every release of the payment region.
- *Outside RG.* Fraud models, pricing and the accounting rules themselves, although the rules, once stated, become invariants.

Agentic and AI-native systems. This class covers support agents, coding agents, research agents, operations agents and teams of agents, together with the people who supervise them. Agents call tools, many of them through MCP [18], run for minutes or days, and act on systems that matter (Figure 25d).

- *What makes it hard.* Steps are non-deterministic and long-running, so they must survive restarts and be resumable. Every tool call needs authority. Content returned by one tool can carry instructions that steer another tool's effect, which is prompt injection, and it can carry data out, which is exfiltration. Each run must leave evidence of what was done. Cost and rate limits must be enforced, and the quality of an agent's work must be evaluated.
- *Today.* Agent loops interleave reasoning and tool calls [361], tool protocols type the calls [18], and graph runtimes make agent workflows durable and interruptible [217]. Frameworks type tool calls but rarely type authority or data flow. Recent designs separate the control path from untrusted data for that reason [74, 89].

In RG three structural facts limit the damage:

- an agent is a principal that proposes;
- each tool is a session with an effect class;
- untrusted content carries a label that privileged sinks reject.

The defence does not depend on the model resisting the injection. It bounds what an injected instruction can cause, although a declassifier or an approver who is deceived can still let content through (Section 23). The same graph gives the agent its context as a traversal with a coverage claim, and records what it did as receipts and claims. Listing 23 shows the shape. An instruction injected into a ticket can make the agent *propose* a refund but not execute one. Only a person holds the grant to approve, and the ticket's text can reach the refund's arguments only through a review declassifier that the same person holds. Runs are durable components whose steps persist their outcomes, so a run interrupted by a deployment resumes rather than repeats its tool calls. Spending and rate limits are budgets on grants. An evaluation of an agent's work is a claim attested by checks and by reviewers, and the accumulated verdicts are the data from which the agent's grants are widened or narrowed (Section 19).

Workflow and process automation. This class covers onboarding, permits, insurance claims, procurement, compliance checks, integrations between businesses and the back office in general (Figure 26e).

- *What makes it hard.* Processes wait for days or weeks on people and on external systems. They have deadlines, escalations and service-level agreements. Integrations with external interfaces and file exchanges are flaky. Regulators ask for evidence of every step. And a process definition changes while thousands of instances are still running.
- *Today.* Workflow engines, process notations such as BPMN, robotic process automation and scheduled jobs each cover part of this; versioning of running instances is usually left to the engine's conventions.
- *With RG.* Steps are durable transitions. A person's decision is a Session with a human surface, so it is a gate like any other call, with a deadline and an escalation that the human profile's laws require (Table 4). Deadlines are timers, and recurring work is a schedule with a misfire policy. External systems are Sessions whose outcomes include

UNKNOWN. A process completes only with receipts, which is the audit evidence regulators ask for, kept as Attests with digests. A change to the process is a patch: running instances stay bound to the revision they started on, and moving them to the new one is an explicit plan, released with draining like any region.
- *Outside RG.* Whether the business rules are the right ones.

Interactive simulation and multiplayer games. This class covers competitive matches, persistent worlds, casual multiplayer, live operations with events and stores, and simulations for training (Figure 26f). Section 4 discussed the engines; here the subject is the system around them.

- *What makes it hard.* Latency budgets are tens of milliseconds. Each match needs one authoritative state and must decide which player sees what. Matchmaking balances skill, latency and capacity across regions. A persistent world is partitioned across servers, and players cross the partitions. Updates must ship during live events, without downtime. Around the game runs a backend of accounts, inventories, stores, leaderboards, anti-cheat and analytics, with every problem of Section 16.
- *Today.* An engine, dedicated game servers with latency compensation [31] and a separate backend stack, joined by convention.
- *With RG.* A match is a keyed room placed near its players. Its tick loop is an exclusive component driven by a fixed-rate timer, and its state updates travel on signal channels at a declared rate. Fast-paced games need unreliable datagrams, which the reference stack does not offer (Section 13), so their rooms run on game-server places. Matchmaking is a solver traversal over players and capacity. A persistent world partitions space into keyed regions, and a player crossing a boundary is an ownership handover with fencing. Updates happen at match boundaries in drain-and-swap mode. The backend is in the same graph: a kill recorded on a leaderboard is a Session with effect identity (Figure 5b), and anti-cheat signals are observations joined to the match that produced them.
- *Outside RG.* Netcode, client prediction, physics and rendering stay inside the engine.

Robotics, IoT and fleets. This class covers warehouse robots, drones, fleets of vehicles, industrial equipment, smart buildings and medical devices (Figure 26g). These are multi-runtime systems with unreliable links and safety-critical change.

- *What makes it hard.* Devices differ in sensors, actuators and compute. Links drop for minutes. Control passes between an autonomy stack, a teleoperator and a failover controller, and must never be held by two at once. A command acts on the physical world, so its outcome may be unknown. Software updates go over the air and must roll back safely, and regulators ask for safety cases that stay true as the software changes.
- *Today.* ROS 2 runs on the robot [231]; fleet managers, update services and safety documents are separate systems.

Three results of this paper apply directly:

- a control handover between an autonomy stack, a teleoperator and a failover controller is the ownership handover of Tables 9 and 34, where a check by the sender is not enough and the receiver must fence;
- a command to a machine is an effect whose timeout means UNKNOWN;
- a behaviour update is a region release with admission and rollback (Listing 6).

The on-robot computation graph is itself a typed port graph of topics, services and actions with quality-of-service profiles [231]. It can be imported as an observed scope, so the fleet and the robot are one graph. Devices are places that offer capabilities, so task allocation is a solver over capability offers. Fleet state that only grows, such as observation logs and sets of visited areas, is merged by monotone traversals that need no coordination while links are down (Proposition 8). A safety claim depends on firmware and parameters, so it goes stale when either changes. Hard real-time control loops and perception models stay inside opaque components.

Data and ML pipelines. This class covers streaming ETL, change data capture, materialized views, feature stores, training pipelines, model registries, analytics and backfills (Figure 26h).

- *What makes it hard.* Computation must be incremental, and data arrives late and out of order, which forces trade-offs between correctness, latency and cost [9]. Schemas evolve while producers and consumers run. Backfills reprocess history. Results must be reproducible: which data trained this model? Retention, deletion and consent apply to every copy. And the features computed for training must equal those computed for serving.
- *Today.* Logs and stream processors [51, 208], incremental engines [46, 236], schedulers and lineage tools, each with its own model of the pipeline.
- *With RG.* The stages of a pipeline are stream components joined by Channels, and each stage maintains its output incrementally with Z-set operators, as in DBSP [46]. Derives records the lineage between the datasets that the stages read and write; like every assertion, it does not execute. Windows and triggers are attributes of the channels between stages. A schema change is a migration patch, and a backfill is a traversal over history that produces a plan. Every model artefact derives from a data snapshot, code and parameters, so the question “which data trained this model” is a query. Retention is an attribute of the dataset, and deletion propagates as in social platforms. Training and serving features are two realizations of one definition, and a claim compares them on samples.
- *Outside RG.* Statistics and the accuracy of the models.

```
val Column = scope("cortical-column") {
    val exc = population("L23-exc", model = AdEx, size = 8_000)
    val inh = population("L23-inh", model = Lif, size = 2_000)
    val drive = population("thalamus", model = Poisson(rate = 8.hz), size = 500)
    projection(exc to inh, synapse = Exponential(tau = 5.ms), connect = FixedProbability(0.1), delay = 1.5.ms)
    projection(inh to exc, synapse = Exponential(tau = 10.ms), connect = FixedProbability(0.2), delay = 1.ms)
    projection(drive to exc, synapse = Static(weight = 0.5), connect = FixedInDegree(50), delay = 2.ms)
    val study = sweep("rates", seeds = 1..20, parameters = grid("g", 4.0, 5.0, 6.0, 7.0, 8.0)) {
        simulation(duration = 10.s, dt = 0.1.ms, checkpointEvery = 1.s, on = capability(Gpu, Nest))
    }
    record(study, exc, Spikes, into = dataset("spikes", label = Internal))
    claim("rates in range", about = exc) { meanRate(between = 2.hz..8.hz) }
}
```

Listing 24: A network study in a proposed neuroscience module of RGL.

Multi-tenant platforms and control planes. This class covers SaaS products, deployments into customers' own clouds, developer platforms and internal platforms, operated as fleets of cells (Figure 27i).

- *What makes it hard.* Tenants must be isolated in data, compute and noisy neighbours. Placement spans customer clouds and regions, under data-residency rules. Fleet upgrades need canaries and per-tenant compatibility. Quotas and billing must be exact. Support engineers sometimes need access to a tenant, and that access must be bounded and recorded.
- *Today.* Kubernetes operators [47], intercloud brokers [360], feature flags and per-tenant configuration files.
- *With RG.* Tenancy rule A6 makes the isolation of data a checked property of the graph; isolation of compute is a placement constraint together with quotas. Cells are places, and data residency is a constraint on the region attribute of places. Bring-your-own-cloud placement is a constrained assignment of components to the places a customer offers, solved over capability offers and certified for constraint satisfaction (Section 23). A fleet upgrade is a sequence of region releases admitted cell by cell, each gated by claims. Quotas are attributes of grants, observations are kept per tenant and element, and access for support is a grant with a time limit whose use leaves receipts.
- *Outside RG.* Pricing and commercial decisions.

Scientific simulation. This class covers brain simulators, climate and earth-system models, multiscale physics and digital twins. Computational neuroscience is our example (Figure 27j). Its studies run on laptops, clusters and supercomputers, often with GPUs, and the software around them is usually scripts that each group writes again. A study often composes models from several groups and scales:

- point-neuron networks in NEST or Brian 2 [142, 321];
- detailed cells in NEURON [171];
- whole-brain models in The Virtual Brain [306];
- several of these in one run, through co-simulation [97].

Model languages such as NeuroML, PyNN and SONATA already separate a model from the simulator that runs it [81, 84, 146]. The science follows computational neuroscience [103]; RG does not choose models or numerics. What it helps with is the software around the science, which is still built by hand for each study:

- *Composition.* A population is an element and a projection is a hyperedge among a source population, a target population and a synapse model, with roles and a delay. A domain module provides these as derived kinds, without a new base kind (Listing 24).
- *Placement.* Simulators are places that offer capabilities: GPUs, MPI ranks, time steps, supported neuron models. A model that no available simulator can run is a finding before the run, not a crash after an hour.
- *Co-simulation.* Simulators exchange spikes on channels with a declared minimum delay. The smallest delay across a cut is the lookahead that lets each side advance without waiting, and a traversal computes it.
- *Long runs.* A run is a durable component that checkpoints; a parameter sweep is a scope factory whose instances are activations with seeds.
- *Provenance.* Every result Derives from a model revision, parameters, seeds, a simulator release and input data, in the spirit of W3C PROV [254] but checked: a figure goes stale when any input changes, like the claims of Figure 24. Workflow managers [94, 252] order the steps; the graph also knows what they mean.
- *Validation and data.* "The model reproduces this firing-rate distribution within this tolerance" is a claim attested by a run and a dataset. Recordings from people carry labels and reach only analyses granted for that purpose.

Neurosymbolic AI. Neurosymbolic systems pair learned models with symbolic reasoning, search or verification [58, 83]. A language model proposes programs that an evaluator scores [301]; a symbolic engine and a language model take turns on a geometry proof [340]; a logic program trains a perception network through its rules [224, 234]. They are hard to build (Figure 27k) because the two halves live in different runtimes and representations, tensors on accelerators and terms

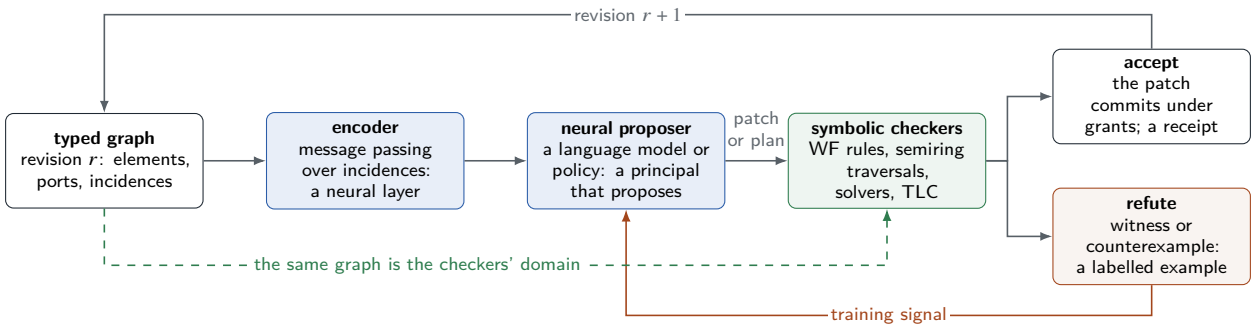


Figure 28: A neurosymbolic loop over the Reaktor graph. Learned components encode the graph and propose changes; symbolic components check them on the same graph, over Boolean, probabilistic or differentiable semirings. Accepted proposals become revisions, and refutations become training data.

on CPUs, joined by glue that no one checks. The Reaktor graph addresses four obstacles, shown in Figure 28:

- *One representation.* The graph is typed, so a graph neural network can embed it: each layer is a message pass over the same incidences as Equation (2) [29, 145], as programs are embedded from their graphs [10]. The same graph is the domain of Datalog-style traversals and model checking. Learned and proven facts about a system sit side by side, distinguished by their evidence: a Derives from a model, or an Attests from a check.
- *Semirings as the bridge.* Traversals already compute over semirings (Section 8.2). With the Boolean semiring a rule computes a proof; with a probabilistic one, a confidence; with a differentiable one, a gradient [156, 224]. One rule then serves the symbolic and the learned half.
- *Checked loops.* A neural proposer, such as a language model writing patches or a policy proposing placements, is a principal that proposes. Rules, solvers and TLC templates accept the proposal or refute it with a witness, on the change path of Figure 15, under grants and effect ceilings.
- *Data with ground truth.* Every patch, finding, counterexample and receipt is a labelled example. A running installation accumulates a dataset of system changes with verdicts, which is the training signal these loops lack today.

Three systems we intend to build follow this pattern: a placement policy learned from receipts whose output a solver certifies; a network that predicts a patch's impact, checked against the exact traversal; and agents that propose refactorings that must pass WF and TLC. In each, the graph does not make the model more accurate. It makes the system around the model feasible to build and safe to run.

What RG does not solve. Several difficulties stay inside opaque components and specialized hosts, where RG places and observes them but does not improve them:

- game netcode and client prediction;
- hard real-time control loops;
- numerical methods inside simulators, and the scientific validity of a model;
- the accuracy of a learned model;
- cryptographic protocols;
- ranking.

18 Programming at the scale of systems

18.1 The scale hypothesis

Software in operation is growing faster than the teams that run it, for three reasons. Agents now write much of the code [185, 282]. One system now spans phones, browsers, edge isolates, servers and data engines. Each of those runtimes brings its own deployment, failure behaviour and debugger. Human attention does not grow with the system. Coordination costs grow faster than team size [41], and a system's structure follows the structure of the communication around it [71]. We state the consequence as a hypothesis.

> **Scale hypothesis.** Past a modest size, the effort that people and agents spend to make one correct change to a multi-runtime system grows with the size of the system when the system is known through its text. It stays nearly flat when the system is known through a typed graph that does three things: it answers questions with coverage, it checks changes before they apply, and it merges independent changes without review.

Figure 29 draws the hypothesis as a shape.

Section 20.4 turns this into the falsifiable hypothesis H7. Two observations motivate it. First, the authors measured their own repositories from agent transcripts, across 962 commits between July and October 2026:

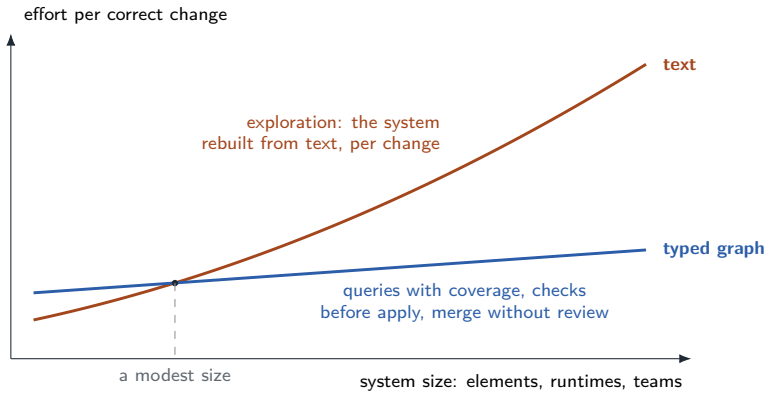


Figure 29: The scale hypothesis (H7), drawn as a shape to be tested, not as data. Below a modest size a typed graph is overhead. Above it, the effort of a change made through text grows with the system, while a typed graph keeps it nearly flat.

- a commit took about 119 model round trips and about 40 million tokens of input context;
- 31–63% of those round trips were exploration and 9–11% were editing;
- agents never called the structural query tools that were available.

The figures come from the local transcripts of two coding agents. A round trip is one model request, context is the sum of the requests' input tokens, and a round trip counts as exploration or editing by the tools it called. We report them as motivation, not as evidence for H7. Second, the survey of Section 20.2 found one system modelled up to seven times inside a single tool, joined by names.

18.2 Raising the level again

Each earlier step in programming raised the unit that people reason about, and moved a class of checking from people to machines:

- compilers took over instruction selection;
- type systems caught whole classes of runtime errors;
- modules enforced the hiding of decisions [279];
- managed runtimes made memory safe;
- declarative infrastructure made fleets converge [47];
- declarative interfaces kept screens consistent with state [242].

Each step was resisted as a loss of control. Each was accepted once its checks proved cheaper than the errors they prevented [96].

The next unit is the system itself: what runs where, talking to what, under whose authority, over which data, with what evidence. The next class of checks is cross-runtime: contracts, effects, authority, information flow and placement. A compiler for one runtime cannot see any of these. RG makes them the subject of types, rules and traversals.

18.3 What changes, dimension by dimension

Table 25 summarises the change.

18.4 People and agents

RG fixes a division of labour that lets the autonomy of agents grow safely.

- *People* own intent, invariants, policy and approval.
- *Agents* explore, propose patches, write code bodies and run analyses.
- *The kernel* checks.
- *Executors* apply.

No principal approves its own proposal. An agent's autonomy is therefore a grant, widened as evidence of its reliability accumulates, rather than a switch. Its context is a traversal with a coverage claim, so it knows what it has not seen. People in the workbench and agents through tools ask the same typed queries, so neither has to reconstruct the system from files.

18.5 Formal methods as a by-product

Lamport asks who builds a house without drawing blueprints [215]. In software the answer has been almost everyone, because blueprints drift from the building. In RG the blueprint is the graph: the building is elaborated from it and reconciled against it. Specifications are assembled from family templates (Equation (1)). Claims are elements whose

Dimension	Text-first practice	With RG	Measure
Understanding	Read code and dashboards; reconstruct the system	Typed queries that state their coverage; views; provenance that explains why (Section 8)	Time to answer; agents' share of exploration
Parallel development	Branches merged as text; conflicts found late	Independent patches merge without review; conflicts enumerable in advance; refs and proposals (Section 9)	Conflicts per change; changes in flight
Review	Textual diffs whose effect must be reconstructed	Typed graph diffs ranked by risk, with computed impact	Review minutes; invalid changes accepted (H6)
Execution	Deploys and configuration written by hand per runtime	Plans compiled from the graph; admission; code push (Sections 9.5 and 13)	Failed deploys; time to ship
Reliability	Retries, timeouts and deduplication coded per call site	Effect identity, fencing, outboxes and durable steps by construction	Lost or duplicated effects under injected faults (H5)
Formal methods	Separate models that drift from the code	Specifications generated from the same graph; claims gate change (Section 15)	Claims with fresh evidence per release
Observability	Logs and spans joined by labels	Observations joined by identity; coverage gaps explicit	Unattributed spans; time to diagnosis
Authority	Permissions in code and configuration	Grants as data, labels on data, no self-approval (A5); tenancy and flow rules (A6, W11)	Ungranted effects found statically

Table 25: Programming dimensions before and with RG.

evidence goes stale when the code changes, and admission requires fresh evidence. Formal reasoning stops being a separate project and becomes a property of each release.

18.6 Risks

Raising the level has four costs.

- *Ceremony.* Over-formalising small features slows them. The common case must stay as easy as ordinary Kotlin.
- *Opacity.* Opaque bodies limit what checks can see, and effect summaries are conservative.
- *Gaming.* Measures such as review minutes invite gaming, so H6 and H7 pair them with the outcomes of injected defects.
- *Understanding.* People who read less code may understand less of it. The graph must therefore explain as well as decide: provenance and why-queries are part of the design, not decoration.

19 Tools, agents and teams on one graph

Every tool that works on software today builds its own model of the system, and each model stops at the edge of a runtime.

- The compiler sees one module, and the IDE sees one project through a language server. The Language Server Protocol standardised how editors talk to such servers, so that one implementation serves many editors [245], but each server still knows one language.
- The profiler sees one process, and the tracer sees spans joined by propagated identifiers but attributed to code by name.
- The test runner sees files and mocks, and the reviewer sees a textual diff.
- An agent sees whatever text it has read.

Each of these tools rebuilds a fragment of the structure that the graph holds. Code property graphs showed the value of merging representations: a graph that combines syntax trees, control flow and data dependence exposes vulnerabilities that none of the three exposes alone [358]. RG does the same for a whole system. With the graph present, a tool is a traversal or an executor, and what it produces (findings, plans, observations and claims) is graph data that other tools read. Figure 30 shows the arrangement, and Table 26 lists what each tool reads and writes.

19.1 Compilers and language tooling

Section 10.7 describes how the compiler uses the graph to optimise. Language tooling gains three further things.

- *Checks across runtimes.* Each compiler checks its own runtime; the graph checks the boundaries between them. A Kotlin/JS Worker calling a JVM service, the Swift shell calling Kotlin/Native, a C++ codec behind a trunk and a SQL schema read by three services are each checked against one contract.
- *Navigation across runtimes.* A language server backed by the graph can go from a call site to the provider that serves it in another runtime and another repository, find every consumer of a channel, and show on hover where a component is placed and under which grants it acts.
- *Diagnostics where the cause is.* A rule that fails on the graph is reported at the source span that created the offending element, so an error about placement or authority appears in the editor, next to the line that caused it.

19.2 Performance engineering

Profilers and tracers today describe one process or one request. Because observations are joined to elements and incidences by identity, the graph turns them into a profile of the system.

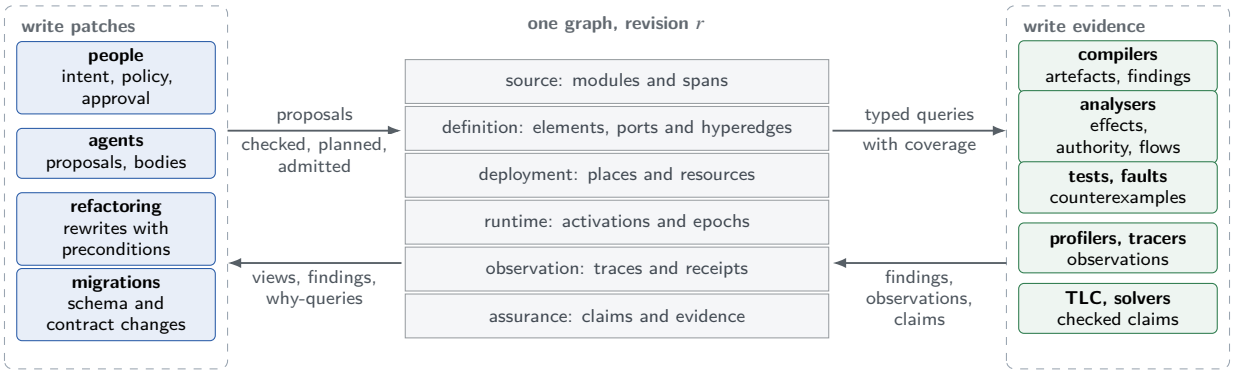


Figure 30: Tools on one graph. People, agents and rewriting tools change the system only through patches, which the kernel checks, plans and admits. Compilers, analysers, tests, profilers and checkers read through typed queries that state their coverage, and write back findings, observations and claims that every other tool can read.

Tool	Its own model today	What it reads from the graph	What it writes back
Compiler	one module	bindings, placements, consumers, profiles: the compilation plan	artefacts with digests; findings
Language server, IDE	one project	the elements behind each source span; connections across runtimes	diagnostics at source spans
Profiler	one process	the latency model; incidences	observations per incidence and revision
Tracer	spans named by service and operation	the instrumentation plan	observations joined by element identity
Test runner	files and mocks	the impact set; contracts; outcome types	evidence that attests claims
Fault injector	random faults	sessions with effects, handovers, outboxes, failure domains	counterexamples as findings
Static analyser	code	effect, authority and flow summaries	findings with witnesses and coverage
Refactoring engine	syntax trees	the preconditions of rewrites	patches
Debugger, incident tools	logs	why-queries, revisions and patches	timelines; runbooks as plans
Review	textual diffs	graph diffs, impact, stale claims	approvals, exercised under grants
Build and CI	build files	the impact traversal	artefacts and evidence
Agent	the text it has read	traversals with coverage	proposals, findings and receipts

Table 26: Tools on one graph: what each reads and what it writes back.

- *Where time goes.* The critical path of a user action, across phone, edge, cloud and database, is a max-plus path problem over observed latencies.
- *What if.* The orders of magnitude of Section 3, calibrated by observed latencies, predict the effect of moving a component before the move. The placement solver uses the same estimates, and observations afterwards confirm or refute the prediction.
- *Which change.* Observations carry revisions, so a regression is attributed to the patch that introduced it, not to the deployment that happened to be nearby.
- *How much load.* The capacity of a single path is a max-min path problem over observed throughput; where paths share a resource, such as a connection pool, capacity is a flow problem over the graph. A load test that shifts live traffic is a plan, observed and reverted like any release.

The same profiles feed profile-guided compilation (Section 10.7), so measurement and optimisation share one representation.

19.3 Testing and fault injection

- *Selection by impact.* Regression test selection by dynamic file dependencies already avoids running unaffected tests [147]. Selection by the impact traversal also follows dependencies that cross runtimes and pass through data.
- *Tests from contracts.* Property-based testing generates inputs from types [62]. Contracts are canonical schemas, so generators come from them directly, and every boundary gets a test on both of its sides.
- *Faults where they matter.* The graph says where faults can hurt: every SESSION with an effect, every handover of an exclusive component, every outbox and every failure domain. A fault plan injects loss, duplication, delay and crashes there, and checks the resulting histories against the TLA$^+$ templates and the declared QoS. Where providers are replaced by models of their families, this is deterministic simulation in the style of FoundationDB [366], aimed

by the graph rather than at random. Against real providers, such as Workers, Durable Objects or Postgres, it is fault injection, not simulation, because their schedules cannot be controlled.
- *Coverage of the system.* Line coverage says little about a distributed system. System coverage counts which incidences, transitions, outcomes (UNKNOWN included) and grants the tests exercised.
- *Replay.* A production failure captured at boundaries becomes a regression test.

19.4 Static analysis and security

Infer showed that compositional analysis scales to very large code bases when procedures are summarised and only changed code is reanalysed [49]. RG's analyses are compositional over the graph in the same way: effect footprints, authority, label flow and tenancy are summaries per element, combined along connections and maintained incrementally under patches (Proposition 7). Analysers that index code bodies, such as CodeQL [21], remain useful inside components. A security review reads grants and flows rather than reconstructing them, and every artefact derives from its sources, so the provenance of a release is a query.

19.5 Refactoring and migration

A refactoring is a transformation that preserves behaviour, guarded by preconditions [273]. In RG a refactoring of a system is a rewrite whose preconditions are checked on the graph, and whose result must refine the original (Corollary 5). A rewrite that adds behaviour is not a refactoring: it states the behaviour it adds, and review approves it as a change.

- *Move a component to another place.* Its Binds that now cross places become Sessions, and the compiler flags every call site that must now handle UNKNOWN. The move adds behaviour (calls that fail or end in UNKNOWN), so it is such a change rather than a refactoring.
- *Split or merge services.* The rewrite reassigns ports and checks that no contract or grant is lost.
- *Extract a region* for code push. The closure computation states what else must move with it.
- *Strengthen a channel's QoS* or *evolve a contract*, with the migration that the change requires.

The graph checks the structure, and agents or people edit the bodies that the rewrite leaves to them.

19.6 Debugging and operations

Provenance makes *why* a query: why a route is unreachable, why a request took two seconds, why a grant was needed (Section 8.2). Revisions make *when* a query: what the system looked like at any time, and which patches separate two times. Centrality on the incidence graph estimates the blast radius of a failing element (Section 3). An incident timeline assembles itself from observations joined by identity, and a runbook is a plan with preconditions rather than a document.

19.7 Review, build and release

Review moves from textual diffs to graph diffs that are ranked by risk and carry their computed impact and the claims they make stale (H6). The build graph is a projection of the system graph, as Bazel's action graph of actions, inputs and outputs is a projection of a build [331]. Artefacts are addressed by digest, and CI runs the impact set rather than everything. Release admission then requires fresh evidence for exactly the digests being released (Section 15).

19.8 AI agents

Studies of coding agents point in one direction: structure helps.

- SWE-agent showed that the interface between agent and computer strongly affects results [359].
- Agentless showed that a fixed pipeline, which localises a fault, repairs it and validates the repair, is competitive with open-ended agents [356].
- CodePlan framed edits across a repository as planning over a dependency graph [25], and RepoGraph showed that a graph of the repository improves agents [276].
- Language models use long contexts unevenly and often miss information in the middle [227].

The graph offers agents what these studies point to, at the level of the whole system.

- *Context with coverage.* A task's context is a traversal: small, typed and relevant, stating what it did not cover. The agent does not need to read files to find out what exists.
- *Localisation as a query.* The elements that a task touches, and their source spans, are computed rather than searched for. Exploration took 31–63% of agent round trips in our measurements (Section 18).
- *Actions with immediate, precise feedback.* An agent proposes a patch, and the kernel answers at once with findings and witnesses. That is a dense, exact signal, unlike a test suite that runs for minutes and reports one failure.
- *Safe parallelism.* Patches to the graph that touch different elements commute (Theorem 10), so structural conflicts appear as soon as two patches exist rather than at merge time, and many agents can work on one system at once. Bodies still merge as text, and patches that commute can still be incompatible in meaning, which checks and tests must catch.
- *Bounded authority.* An agent is a principal with grants and effect ceilings, and labels bound what injected content can cause (Section 17.1).

- *An interface designed for agents.* Query, propose, check, plan and observe are typed operations, exposed as tools in the Model Context Protocol [18].

19.9 People and agents together

Section 18 fixes the division of labour: people own intent, invariants, policy and approval; agents explore, propose and write bodies; the kernel checks; executors apply. Six patterns of collaboration follow from it.

1. *Intent first.* A person states intent as claims and invariants, and agents propose patches that must satisfy them.
2. *Designer and programmer.* As in a game studio, a person composes features in graph-owned scopes in the workbench, as a designer composes Blueprints, while agents write and maintain the Kotlin bodies behind the ports, as programmers maintain the C++ behind reflection (Section 4). The roles can also be reversed.
3. *Review by exception.* The kernel checks everything mechanical, and people review what the risk ranking surfaces: changes of authority, new external effects, declassifications and stale claims.
4. *Teams of agents with roles.* A planner, an implementer, a tester and a reviewer are distinct principals with distinct grants. Because no principal approves its own proposal (A5), review by a second principal is enforced rather than customary. A5 alone does not make review independent: an operator could mint a second agent with the same model and the same inputs. Approval grants above an effect class are therefore held by a person, or by a principal with its own delegation chain and a different model.
5. *Earned autonomy.* An agent's grant widens as receipts and accepted patches accumulate, and narrows after a bad change. Autonomy is evidence-based rather than a switch.
6. *Explanation on demand.* Programmers who work with code-generating models alternate between accelerating what they already intend and exploring what they do not yet know [27]. The graph supports both: typed completion of edits for the first, why-queries for the second, so that people stay in control without reading all the code.

19.10 Risks

Moving tools onto one graph creates new risks.

- *Automation bias.* People may approve what the ranking does not flag; H6 and H7 measure accepted defects, not only speed.
- *Deskilling.* People who read less code may understand less of it, so explanation is part of the design.
- *A valuable target.* The graph is a complete map of the system, so it must be protected by the same grants and labels as the data it describes.
- *A dependency.* If the kernel is unavailable, tools fall back to their own models; the graph must degrade to a cache, not to an outage.

20 Evaluation

RG is a design, so this section reports what can be measured before implementation. It applies the type system to the existing design corpus and to a survey of a production platform, checks two activation protocols exhaustively within bounds, and states the hypotheses that will accept or reject the implementation. Requirements coverage is in Tables 1 and 20.

20.1 The 2026 design corpus under Σ_{std}

Method. The corpus is every definition graph published with the Reaktor documentation on 7 October 2026: 140 graphs comprising one example per kind of the 2026 catalogue (100), 30 scenario designs (mobile, web, distributed, capabilities, workflows, games, IoT, robotics), 5 nested-graph examples and 5 view examples. Together they hold 750 nodes, 1 420 ports, 648 relations and 1 420 incidences. 542 relations are binary, 88 ternary and 18 quaternary. A script, included with the paper, maps each relation to a family of Σ_{std}. The relation kind decides, except for generic value flows, where the port profile decides. The script then normalises role names and applies the checks of Section 6.5 that are decidable on the current IR: roles (W3), nets (W4), contracts (W5), layers (W6) and locality (A1). For locality it uses region labels as a proxy for places.

Results. Table 27 supports four claims. First, the fourteen families and the containment forest suffice for the corpus. Second, a separate relation kind is redundant. Among the connections whose ports imply a connection family, the family follows from the port profile in 95.4%. Leaving out the value flows, whose family the script takes from the profile and which therefore agree by construction, the rate is 93.0%; counting the 31 connections whose ports imply no connection family as disagreements, it is 89.3%. This is why RG types ports by family and role and derives the family of the hyperedge. The remaining 21 cases are genuine ambiguities, for example an "exclusive binding" over request–response ports, which RG forces the author to resolve. Third, the vocabulary contracts sharply. Role names fall from 123 to 31; polarity becomes derivable, with no contradictions; and almost a quarter of all ports disappear, because assertions attach to elements directly instead of to dedicated inspection ports. Fourth, the static checks find modelling errors that the structural validator accepts. Among them are bindings across hosts that hide a network boundary; grants that conflate

Measure	Result
Relations expressed in Σ_{std}	648 of 648: 486 connections (Session 185, Bind 141, Channel 98, Transition 62), 142 assertions, 14 containment links, 6 stream operators
Connections whose family follows from the port profile	434 of 455 (95.4%); 277 of 298 (93.0%) without the value flows, which agree by construction; the other 31 run over ports that imply no connection family
Distinct role names (role slots)	123 → 31 (192 → 37)
Incidence polarity versus role-derived polarity	977 agree, 19 unspecified and now derived, 0 contradict
Ports that existed only to carry metadata	346 of 1 420 (24.4%), replaced by assertions on elements
Ports used in more than one relation	5 of 1 420: the net discipline already holds for 99.6%
Region labels that denote places, not scopes	267 of 422 (63%)
The 2026 "value flow" kind, split by law	Bind 94, Channel 47, Session 16, Observes 10, Derives 3, Grant 1
Findings that the current validator accepts and Σ_{std} rejects	
Bind across two host lanes, the catalogue's places (A1)	21
Connection spanning layers (W6), of which a channel reified as a node	78, of which 28
Roles that are realizations or solvers (broker, planner, publisher, …)	28
Grant whose "principal" slot holds the policy that issues it	10
Cancellation modelled as control precedence	9
Route binding encoded as a boundary export	5
Join without an explicit continuation (W3)	3

Table 27: The 140 published design graphs under Σ_{std} (`anc/rg_catalog_eval.py`).

Measure	Result
Features expressible with the fourteen kinds and fourteen families	283 of 283
Entries that RG improves materially, clearly, or marginally	76, 69 and 5 of 150; about 30 further features are not improved: computation inside components, drawing, external platforms and human decisions
Additions the mapping required	11, none a new base kind or family: labels (W11), tenancy (A6), one effect lattice, schedules, traversals compiled into stores, dataset invariants, receipts bound to payloads, a render-plan protocol for pushed UI, refs and proposals, references between any elements, quotas on grants
Recurring patterns (each found in several surveyed systems)	
One system, many models	up to 7 models of one system in one tool, joined by names; deployment topology in 7 hand-kept places
Identity that does not survive a run	4 observation paths with incompatible identity; joins by label
Effects lost or misreported	post-commit effects without an outbox; expired deadlines reported as failures
Actions without principals	approvers as free strings; self-approval possible on every tooling path
Authority and capability in code	31 process-global service slots; 4 call styles between runtimes; 6 effect and truth vocabularies

Table 28: A survey of 283 features of Reaktor and five systems built on its current runtime.

the issuer with the principal; cancellation encoded as precedence; and brokers and planners recorded as participants of the relations they realize or compute. The 2026 "value flow" kind alone splits into six families with different laws.

Threats. The mapping is ours, and the corpus was written by the same project's authors and agents. Region labels only approximate places. The checks are static and cover part of WF. The study shows that Σ_{std} is sufficient and sharper for the designs Reaktor already has; it does not show that it is sufficient for systems not yet designed. H1 below tests that.

20.2 A survey of a production platform

Method. Seven parallel read-only surveys covered Reaktor's modules, its tools and five systems built on its current runtime:

- a social messaging application;
- a knowledge-and-work system;
- the developer workbench;
- a business-to-business commerce prototype;
- a back-office automation system.

The surveys were carried out by LLM-based code-reading agents under the authors' direction. Every recorded fact cites a file and line or a section of a document, and the authors spot-checked the key claims. They produced 283 feature records, which we grouped into 150 entries and mapped onto Σ_{std} (Table 28).

Results. The survey supports H1 on a broader sample than the corpus, drawn from running systems rather than designs. It also changed the design. The eleven additions in Table 28 are now part of the design, in Sections 6 to 9, and each is a facet, a derived kind, a rule or a kernel feature.

Threats. The mapping and the grouping are ours. The surveys read code but ran nothing. The systems were written by this project's authors and agents.

20.3 Bounded protocol models

Table 9 explores two protocols that activation and code push rely on. The first is ownership handover between activations. Without fencing at the receiving authority, a three-step trace accepts a write from an obsolete owner. A sender that checks its lease before every write fails on the same trace. This is the counterexample of Reaktor's language design report [294], reproduced here with a different encoding, which changes the state counts but not the trace. With fencing, no violation exists within bounds of up to three owners, two writes each and three handovers (4 555 states). The second is effect identity under loss, duplication and retry. Without deduplication by effect identity, an effect is applied twice. With deduplication, a client that treats a timeout as "not applied" still reaches a false conclusion. Treating it as UNKNOWN is sound. These results are the reason for two design decisions: the Outcome type of Session calls, and the epoch check inside `atomic(store)`.

20.4 Hypotheses for the implementation

Each hypothesis has a rejection criterion. Baselines are the current runtime's kernel, Koin and Dagger for injection, Compose Navigation for routing, direct Ktor and Hono services, Temporal for durable workflows [329], and Expo or React Native update services for code push. For the hypotheses about compilation, tools and agents (H8–H15), the baselines are the same code built without a compilation plan but with the best current build-time tools (Dagger or Hilt with R8 on Android [150, 152]), full rebuilds, the test suite run in full, random fault injection, and the same agents with file-based tools only.

H1 Coverage. The graphs of three production systems on the current runtime (both roots of a social application, a knowledge-and-work system and the workbench) convert to Σ_{std} with no new base kind or family. *Reject if* any requires a kernel change.

H2 Defects. The RG checks report the known defects of the current runtime (duplicate providers of one port, unwired consumers, spans attributed by label, calls between workers modelled as local bindings), with at most 10% false positives on manual review. *Reject if* a known defect is missed.

H3 Kernel cost. Snapshot below 1 µs. Injection plan below 1 ms for an application of about 300 ports. Traversal throughput at least 10^7 incidences per second on the JVM and 10^6 in Hermes. Local Bind call overhead at most 5% over a direct Kotlin call. *Reject if* any bound is missed by a factor of two.

H4 Incrementality. Maintaining findings and plans under patches that touch at most 1% of elements costs at most 5% of recomputation. *Reject if* the median cost exceeds 20% of recomputation.

H5 Code push. On a mid-range Android device, a stateless region pushed in next-entry mode activates within 300 ms. Across 1 000 releases with injected faults (release during a session, lost acknowledgement, crash after commit), no call or effect is lost or duplicated. *Reject on* any loss or duplication.

H6 Review. In a matched study following the design of [288], developers and agents that review graph patches accept fewer invalid changes, and find injected defects faster, than with textual diffs. *Reject if* neither difference is significant at the 5% level.

H7 Scale. The study of H6 is repeated on systems of about 10^2, 10^3 and 10^4 elements. With RG, review minutes and agent round trips per accepted change grow sublinearly in system size, and exploration falls below 20% of agent round trips, from the measured 31–63% (Section 18). *Reject if* effort per change grows with system size at the same rate as with textual diffs.

H8 Compilation. Compiling release regions with a compilation plan (static injection, devirtualisation, specialised sessions and codecs, and initialisation at build time; Section 10.7) reduces the cold start of an edge Worker and of an Android process by at least 30%, and bundle size by at least 15%, relative to the same code with injection at run time and reflective registries. Against Dagger or Hilt with R8, it must not be slower or larger. Results on the conformance suite are identical. *Reject if* either reduction is below half its target, if it loses to Dagger or Hilt with R8, or if any conformance vector differs between the compiled and the interpreted region.

H9 Incremental builds. Replaying the recorded commits of the surveyed systems, the impact set of each graph diff rebuilds at most 20% of the artefacts that a full build rebuilds, at the median. *Reject if* any artefact whose inputs or assumptions changed is not rebuilt, which is checked by a full rebuild and a comparison of digests, or if the median exceeds 40%.

H10 Testing and faults. Over 500 historical commits, test selection by graph impact runs at most 30% of the tests at the median and never skips a test that failed. In the chat pilot with seeded defects, fault plans generated from the graph (Section 19) find at least as many distinct defects as random fault injection with the same budget. *Reject if* a failing test is skipped, or if the generated plans find fewer seeded defects than random injection.

H11 Agents. On change tasks drawn from the surveyed systems, the same agent resolves at least 10 percentage points more tasks with graph tools (traversals with coverage, patch operations and the kernel's findings) than with file tools alone. Resolution is the primary endpoint, tested one-sided at the 5% level on a number of tasks fixed in advance by a power analysis for that difference; round trips, expected to fall by at least 40%, are secondary. *Reject if* resolution does not improve significantly, whatever happens to round trips.

H12 People and agents together. A 2×2 study crosses who works (people alone, or people with agents, following the patterns of Section 19) with what they work on (text, or the graph). Its prediction is an interaction: the graph reduces accepted invalid changes more for people with agents than for people alone, at equal or lower time per accepted change. *Reject if* the cell of people with agents on the graph is not better than every other cell.

H13 The twenty problems. For each problem of Section 16, defects seeded in the chat pilot by people outside the design team, drawn from the incidents reported in the cited studies, are either rejected before activation or reported in production with a witness. *Reject if* more than two of the twenty problems go unreported.

H14 Observability. On managed connections, at least 99% of spans, logs and metrics are joined to an element by identity, against a baseline measured on the current runtime before the change. On injected incidents, the time to identify the responsible patch falls by at least 30%. *Reject if* fewer than 95% are joined, or if the time to identify the patch falls by less than 15%.

H15 Composition without code. After two hours of training, product designers who do not write Kotlin compose and change features in graph-owned scopes in the workbench, as designers compose Blueprints (Section 4). *Reject if* defects found after admission are more than twice as frequent in their patches as in programmers' patches for comparable tasks.

21 Related work

Graph models and their algebra. Bigraphs separate a place graph from a link graph and give both an algebra and reactive rewriting [247]. RG's containment forest and incidences have the same split, and add typed ports, role-labelled ordered incidences, layers and attributes. Port graphs and their attributed hierarchical form [108, 133] contribute ports and strategic rewriting, but their edges are binary. Algebraic graph transformation [102, 159, 210] supplies the DPO theory behind Section 9. Attributed C-sets and their rewriting library [42, 280] supply both the formal presentation and a demonstration that it runs efficiently. TypeQL types polymorphic entity–relation–attribute data with reified relations [99], the closest database analogue of hyperedges with roles. Hypergraph categories, string-diagram rewriting, decorated and structured cospans, wiring diagrams and Span(Graph) [23, 35, 135, 136, 193, 315] supply the compositional semantics. We take theorems from these works rather than vocabulary. The contribution here is an engineered signature, an identity and transaction model, and a layered schema on which those theorems apply to production software.

Programming distributed systems. Actors [7, 169], Erlang/OTP [20] and Orleans' virtual actors [34] map to the exclusive profile, the mailbox channel and keyed realization; supervision with restarts is a runtime service that the design specifies but this paper does not show. Multiparty session types and choreographies [50, 173, 253] give deadlock freedom by construction; RG uses them as the semantics of Session. Multitier languages [72, 351], X10 [57], Ray [256], Service Weaver [143], Dapr [333], Temporal [329] and Hydro [59] each make one program span machines along one dimension: tiers, places, tasks, deployment, sidecar building blocks, durable steps or dataflow. RG is not a competing language for one tier. It is the substrate in which such programs are one projection. Placement, data, authority, evidence and change are first-class, and Hydro-style monotone analysis is Proposition 8.

Infrastructure from code and durable execution. Infrastructure-from-code tools derive cloud resources from application code: Encore by static analysis, Wing through preflight and inflight code, Nitric through declared resources and, earlier, Klotho through annotations [107, 204, 267, 353]. Kotless did the same for Kotlin [328], and an empirical study compares language-based and library-based approaches [312]. Ballerina shows network interactions as sequence diagrams and generates deployment artefacts from code [350, 354]. RG's deployment layer is generated in the same spirit (Section 13), but from a graph that also carries contracts, authority, labels and evidence. Temporal, Restate and DBOS make steps durable, the last by checkpointing workflows into Postgres [223, 299, 313, 329]; RG uses such engines as realizations of durable transitions. The term *multi-runtime* is also used for sidecar architectures in which a separate process supplies distributed primitives to business logic, as Dapr does [175, 333]. In this paper it means a system whose parts run on different runtimes, which such primitives can serve.

Architecture description languages. ADLs model components, connectors and configurations explicitly [237]. Darwin composes components through provided and required services, with semantics in the π-calculus [233]. Wright types connectors by role protocols and checks ports against roles [11]. Acme gives ADLs a common interchange core [141], and AADL models embedded systems with modes for reconfiguration [132]. Kramer and Magee's quiescence is the classic condition for safe dynamic change [207]; region closure and draining play its part in Listing 6. RG differs in treating the architecture as the authoritative, executable graph of the running system rather than a description beside it, and in adding layers, data, authority and evidence.

UI, reactivity and code push. React and the Compose runtime reconcile keyed trees [149, 242]; functional reactive programming and Elm [80, 106] give the dataflow view. Zipline and Redwood ship Kotlin/JS behind typed interfaces into native hosts [52, 53]. Dynamic software updating, version consistency and Erlang release handling [129, 170, 262], together with the update framework [305], define safe replacement. RG composes these with a closure computation and Corollary 5.

Infrastructure. Borg and Kubernetes [47, 347] established typed desired state with level-triggered controllers, and custom resources, operators and admission webhooks made the schema and its checks open [335, 337, 338]. Their relations are labels and selectors, and they offer no multi-object transactions. Nix [98] and Unison [60] established content-addressed artefacts and code. Ptolemy's heterogeneous models of computation [104], MLIR's dialects [218] and typed structure editors [271, 296] inform profiles, open signatures and typed holes.

Game engines and visual programming. Unreal Engine composes a C++ core through reflection and typed graph languages: Blueprints, which succeeded Kismet, and graphs for materials, animation and AI [110, 120, 124, 126, 128]. Its runtime adds data-oriented entities, replication with an authoritative server, and streaming of a partitioned world [119, 125, 127]. Unity offers visual scripting and a data-oriented entity system [341, 342], and Godot removed its visual scripting language in version 4 [225]. Visual dataflow languages are older: LabVIEW's graphical language G shipped in 1986 [206], and surveys classify visual languages and trace dataflow programming from the 1970s [187, 260]. Underneath lie Kahn's determinate networks of processes communicating through FIFO queues [191] and synchronous dataflow, whose fixed rates allow static schedules [220]. Section 4 explains what RG takes from the engines. Relative to dataflow languages, it adds placement across runtimes, families with different laws, and change as a first-class operation.

Statecharts, processes and agent graphs. Statecharts [163], their standard execution in SCXML [28] and their implementation with actors in XState [320] supply the hierarchy, orthogonal regions and history of navigation, while the Transition family fires as a Petri net (Table 7). BPMN [270] and the workflow patterns [343] describe business processes; in RG a process diagram is a view of a durable Transition net whose gates are sessions. LangGraph makes the workflows of agents durable graphs with human interruption [217]. RG connects all of these to the system they act on: sessions, channels, grants and evidence.

Model-driven engineering and low code. Model-driven engineering uses domain-specific models and generators that synthesise code from them [307], and low-code platforms share many of its ideas [93]. Architecture description languages, discussed above, model structure in the same spirit. RG differs in where the model lives: it is elaborated from the code and reconciled with the running system, so it does not drift from the code's declarations or from what runs, although a body can still diverge from its contract. It also carries data, authority and evidence as well as structure.

Interface definition languages. Protocol Buffers, OpenAPI and Smithy describe interfaces and generate code from them [15, 151, 274]. Protocol Buffers also states rules for evolving a schema: fields are identified by numbers that must never change. RG treats such definitions as sources of contracts. A contract's fingerprint is a digest of its canonical form, and compatibility between two fingerprints is computed rather than declared (Section 6).

Dataflow systems and placement. TensorFlow places the operations of a dataflow graph on devices, partitions the graph per device, and replaces every edge that crosses devices with send and receive operations, invisibly to the program [3]. RG partitions by place in the same way but makes the boundary visible: a Bind never crosses a place, and an edge that does is a Session or a Channel whose failures the program handles. FlumeJava defers evaluation, optimises the resulting plan and fuses stages before running it [55]. The Dataflow model treats data as unbounded and makes the trade-offs among correctness, latency and cost explicit [9]. Halide separates an algorithm from its schedule [285]. RG lifts these ideas from one data-parallel program to a system of many runtimes.

Compilers and whole-program optimisation. Class hierarchy analysis binds dynamic calls statically when the whole program is known [88], and dynamic deoptimisation discards optimised code whose assumptions an edit has broken [172]. Partial evaluation specialises an interpreter to a program [188], which Truffle and Graal turn into fast language runtimes [355]. GraalVM Native Image initialises a closed world at build time [352], ThinLTO makes cross-module optimisation incremental through summaries [186], and profiles guide code layout [283]. Stream fusion, escape analysis and automatic batching of data fetches remove intermediate structures, copies and round trips [61, 77, 235]. R8 shrinks and optimises Android apps from their entry points [152], Dagger resolves injection at compile time [150], and Service Weaver chooses between local and remote calls at deployment [143]. RGL reuses these techniques (Section 10.7). Its contribution is a domain in which they are sound across runtimes: the elements of a release region admitted together on one host, whose facts admission guarantees.

Graph and analytical stores. Property-graph query languages include Cypher [139] and the ISO standard GQL [176], and their foundations are surveyed in [17]. Columnar storage and vectorised execution made analytics over billions of rows interactive [36, 238, 322]; ClickHouse is a recent open-source example [308]. RG uses such stores as realizations of Stores and compiles traversals into their languages (Section 8.3).

Failures of distributed systems. Studies of production failures locate where systems break: in error handling [362], upgrades [158, 365], partitions [13], partial and gray failures [174, 229], metastable states [39], configuration [326, 357], interactions between systems [327] and emergent behaviour [249]. Practice supplies the mitigations: idempotent interfaces, retries with budgets, logs instead of dual writes, staged schema change, tracing and live load tests [40, 131, 198, 284, 311, 345], and independent testing shows how often guarantees fail anyway [196]. Section 16 draws its twenty problems from this literature. What RG adds is one place where the mitigations are stated as types and rules and checked before activation.

Tools and agents. Code property graphs merge syntax, control flow and data dependence for analysis [358]; the Language Server Protocol decouples editors from languages [245]; regression test selection, property-based testing, compositional static analysis and behaviour-preserving refactoring are mature [49, 62, 147, 273]. For agents, the interface between agent and computer matters [359], simple pipelines of localisation and repair compete with open-ended agents [356], graphs of the repository help [25, 276], long contexts are used unevenly [227], and people alternate between accelerating and exploring when they work with code models [27]. These tools and agents each work on one representation of one program. Section 19 places them on one graph of the whole system.

Code as data, incrementality and assurance. CodeQL and Datalog [4, 21] index code after the fact. In RG the graph is *elaborated* from source and is authoritative for definitions. Adapton, differential dataflow, DBSP, build systems and incremental attribute grammars [46, 161, 236, 251, 297] make derived facts cheap to maintain. TLA$^+$ at AWS, Alloy, P, IronFleet and FoundationDB's simulation testing [92, 165, 178, 264, 366] are the industrial models for Section 15.

Networks and learning. Multilayer and heterogeneous information networks [26, 197, 265, 323] provide the vocabulary and analyses (centrality for blast radius, community detection for proposing regions, percolation for failure domains) that RG can apply to systems. Message-passing networks and Transformers [29, 145, 344] motivate one primitive over typed incidences.

Systems by class. The strongest tools for the classes of Section 17 each own one guarantee:

- CRDTs and local-first software own convergence [202, 310];
- logs and stream processors own ordered, replayable data [51, 208];
- ROS 2 owns typed robot computation graphs with quality-of-service profiles [231];
- game servers own latency compensation [31];
- agent loops and tool protocols own reasoning and tool calling [18, 361];
- intercloud brokers own placement across providers [360];
- neural simulators, model languages and co-simulation frameworks own the numerics of brain models [97, 142, 146, 171];
- neurosymbolic languages own differentiable reasoning [224, 234].

RG treats each as a realization or a component profile and adds what none provides: the guarantees that must hold across tools.

Information flow and authority. Lattice-based information flow [91] and decentralized labels [259] are the basis of RG's labels (W11). Capability systems and relationship-based authorization [246, 277] inform Grant. For agents, CaMeL separates a privileged planner from a quarantined model that reads untrusted data and enforces policies at tool calls [89], and FIDES tracks confidentiality and integrity labels in the planner [74]. RG states the same separation as grants and labels over the whole system graph.

Edge platforms. Cloudflare's Durable Objects realize keyed single-writer places with co-located storage [64], and Orleans showed how such places become a programming model [34]. RG keeps the platform's uniqueness and placement, and adds typed protocols, fencing at the external stores that check epochs, and the rest of the system around them.

Programming at scale. Section 18 builds on four arguments: Brooks on coordination cost [41], Conway on structure following communication [71], Parnas on hiding decisions behind modules [279], and Lamport on specifying before building [215]. Benchmarks of agents on real repositories [185] and studies of assisted programming [282] motivate the scale hypothesis. We do not know of prior work that states it as a falsifiable claim about effort per change.

Reaktor's lineage. The engine preprint [288] defined the port multihypergraph, interaction profiles, a live-update model and a reusable evaluation design. The RGL report [294] defined the distributed abstract machine, the operations, the Hermes host profile and region releases. This paper subsumes both. It adds layers, a minimal family signature, compositional semantics, a theory of traversals, a DPO change model, a kernel, a reference realization and an evaluation on the corpus and on a production platform.

22 Implementation plan

The plan builds identity and the kernel first. It then moves services onto them in the order that retires the most workarounds in the current runtime, with a production chat feature as the pilot throughout. Releases of the surrounding platform are

	typed multi-party relations	nesting, mounting	layers	multi-runtime placement	transactional change	formal semantics	UI	code push
Kubernetes	–	∘	∘	✓	–	–	–	∘
React, Compose	–	✓	∘	–	∘	∘	✓	–
Erlang/OTP, Orleans	∘	∘	–	✓	–	∘	–	∘
Service Weaver, Dapr	–	–	∘	✓	–	–	–	–
Temporal	–	∘	∘	∘	∘	∘	–	–
Hydro	∘	–	∘	✓	–	✓	–	–
Choreographies	✓	∘	–	∘	–	✓	–	–
Ptolemy II	∘	✓	–	–	–	✓	–	–
XState, SCXML	–	✓	–	–	–	∘	∘	–
TensorFlow	∘	∘	–	✓	–	–	–	–
Unreal Blueprints	–	✓	–	–	–	–	✓	–
Zipline, Redwood	–	∘	–	∘	–	–	✓	✓
Encore, Wing	–	–	∘	✓	–	–	–	–
Darwin, AADL	∘	✓	–	∘	–	✓	–	∘
Bigraphs	∘	✓	–	–	✓	✓	–	–
Reaktor (current runtime)	–	✓	–	∘	–	–	✓	–
RG (design)	✓	✓	✓	✓	✓	✓	✓	✓

Table 29: Primary abstractions of related systems, in our reading (✓ central, ∘ partial or indirect, – absent). The table compares scope, not quality: each system is deeper than RG along its own axis. The RG row describes the design in this paper, not a running implementation.

not blocked: the current runtime keeps running behind compatibility facades until each service has moved. Each quarter ends with acceptance criteria drawn from Section 20.4.

Quarter	Deliverables	Acceptance
Q4 2026	Kernel v0 in Kotlin common with no dependencies: C-set tables, stable identifiers, digests, patches, refs and proposals, $\mathsf{WF}_{\mathsf{static}}$, and Σ_{std} as data, including labels, tenancy and the effect lattice. Read-only projections of the current runtime's graphs, the documentation graph and the deployment inventory. Principals, grants and the effect lattice on the agent and tooling change path. Corpus converter; Alloy model of the judgments.	All 140 corpus graphs convert; projections round-trip with identifiers stable across runs; no path lets a requester approve its own plan; property tests of Proposition 9
Q1 2027	Traversal engine (folds, semirings, lattice Datalog, Z-set incrementality), with compilation into Postgres and graph stores. Standard passes: injection, routes, lifecycle plans, effects and flows, capability, closure, lints. Session outcomes and contract fingerprints beneath the current ports, through a facade. Channels with `atomic(store)` and an outbox on Cloudflare Queues. Graph diffs in the workbench. RGP Connect, Query and Watch, with Z-set deltas and reconciliation by digest, serving the workbench and agent tools. Test selection by impact and fault plans generated from the graph. Chat pilot, seeded with the defects of Section 16.	H2; H3 on the JVM; H4
Q2 2027	Durable transitions on Workflows and Durable Object alarms; schedules. The deployment layer generated for Cloudflare, Google Cloud and Postgres. RGL definitions and behaviour on Android, iOS, the JVM and JavaScript. Hermes host profile. TLA$^+$ templates generated from the graph, starting from the five of Table 34, with TLC in CI. The K2 plugin with compilation plans and incremental builds from graph diffs. RGP Change and Release, with effect identity and `lookup`. Observations joined by identity in ClickHouse.	H3 in Hermes; H14; truthful-state conditions (i)–(ii) in CI
Q3 2027	Keyed actors with fencing beyond Durable Objects, on JVM shards. Placement solver. Code push of stateless regions, first to owned desktop and server hosts, then next-entry on Android. Lean proofs of preservation and determinism. RGL Core specification and conformance suite, including RGP frame vectors; federation between authorities. Agent tools over the graph and the graph-owned editor in the workbench. Paper revision with measurements.	H1; H5; H8, once the conformance suite exists; H9 and H10 on the converted systems; H13; studies for H6, H7, H11, H12 and H15; platform policy review for iOS updates

Table 30: Twelve-month plan.

23 Limitations and open problems

- **Opaque bodies.** Arbitrary Kotlin is Turing-complete and stays opaque. Effect summaries are conservative, and purity cannot be verified in general. Routing all platform access through capability ports narrows the gap, but reflection and global state can still escape. The host profile must restrict them where guarantees are claimed.
- **Liveness and performance.** Theorem 3 composes safety. Liveness, latency and cost need assume–guarantee reasoning and measurement; they do not compose for free.
- **Placement** is NP-hard in general. Solvers are heuristic, and only their constraint satisfaction is certified.
- **The signature evolves.** Kinds and families are data, so changing them needs migrations at the meta level. We expect the C-set machinery (Δ, Σ, Π along schema maps) to serve, but this is not yet designed.

- **Scale.** Graphs of millions of elements need partitioned authorities. Non-monotone queries across partitions then pay for consistent cuts.
- **Provider qualification.** Each realization must be qualified per provider and version. A provider's change can silently weaken a guarantee unless the qualification suite runs continuously.
- **Labels.** Information-flow labels are only as strong as the discipline of declassifiers and the visibility that opaque bodies allow.
- **Platform policy.** Code push is constrained by store rules that the graph cannot decide; iOS may admit only interpreted behaviour updates.
- **Mechanisation.** The many-sorted, role-asymmetric form of the cospan correspondence in Theorem 3 is standard but not yet mechanised, and attribute rewriting relies on [42].

24 Conclusion

Kubernetes made a fleet of machines programmable by giving it one typed object model and controllers. We have argued that the analogous step for applications that span phones, browsers, edge isolates, servers, databases and model hosts is a typed, layered port hypergraph. Traversals of it are the platform's services, patches to it are its changes, and its composition is the system's behaviour. Reaktor Graph makes this precise with fourteen base kinds, four connection and ten assertion families, a compositional semantics, a theory of traversals and a DPO change model, all realized as a columnar kernel and, concretely, in Kotlin on Cloudflare, Google Cloud and Postgres. RGL makes it usable from Kotlin, on every host, and in Hermes for code push. Reaktor already holds the ingredients: a runtime in production, a hypergraph IR, a design corpus of 140 graphs and a semantic constitution. The plan above assembles them. If it succeeds, people and agents will build, change and verify systems through one graph instead of through the scattered text that describes them today, and the effort of a correct change will stop growing with the size of the system.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3): 507–535, 1995.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] M. Acton. Data-oriented design and C++. Keynote, CppCon, 2014.

[6] H. Adkins, B. Beyer, P. Blankinship, P. Lewandowski, A. Oprea, and A. Stubblefield. *Building Secure and Reliable Systems*. O'Reilly Media, 2020.

[7] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[8] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In D. Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 452–476. Springer, 2006.

[9] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow Model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[10] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018.

[11] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[12] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[13] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 51–68, 2018.

[14] Amazon Web Services. Amazon Aurora PostgreSQL Limitless Database is now generally available. AWS News Blog, https://aws.amazon.com/blogs/aws/amazon-aurora-postgresql-limitless-database-is-now-generally-available/, 2024. Accessed October 2026.

[15] Amazon Web Services. Smithy 2.0 specification. https://smithy.io/2.0/spec/index.html, 2026. Accessed October 2026.

[16] T. J. Ameloot, F. Neven, and J. V. den Bussche. Relational transducers for declarative networking. *Journal of the ACM*, 60(2): 15:1–15:38, 2013.

[17] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.

[18] Anthropic. Model Context Protocol specification. https://modelcontextprotocol.io/specification, 2025. Accessed October 2026.

[19] Apple. App review guidelines, sections 2.5.2 and 4.7. https://developer.apple.com/app-store/review/guidelines/, 2025. Accessed October 2026.

[20] J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology (KTH), Stockholm, 2003.

[21] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer. QL: Object-oriented queries on relational data. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 56 of *LIPIcs*, pages 2:1–2:25, 2016.

[22] J. Avigad, L. de Moura, S. Kong, and S. Ullrich. Theorem proving in Lean 4. https://lean-lang.org/theorem_proving_in_lean4/, 2026. Online book. Accessed October 2026.

[23] J. C. Baez and K. Courser. Structured cospans. *Theory and Applications of Categories*, 35(48):1771–1822, 2020.

[24] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[25] R. Bairi, A. Sonwane, A. Kanade, V. D. C., A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet. CodePlan: Repository-level coding using LLMs and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698, 2024.

[26] A.-L. Barabási. *Network Science*. Cambridge University Press, 2016.

[27] S. Barke, M. B. James, and N. Polikarpova. Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111, 2023.

[28] J. Barnett, R. Akolkar, R. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. McGlashan, T. Lager, M. Helbing, R. Hosn, T. V. Raman, K. Reifenrath, N. Rosenthal, and J. Roxendal. State chart XML (SCXML): State machine notation for control abstraction. W3C Recommendation, https://www.w3.org/TR/2015/REC-scxml-20150901/, 2015. W3C Recommendation, 1 September 2015.

[29] P. W. Battaglia et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[30] C. Berge. *Hypergraphs: Combinatorics of Finite Sets*. North-Holland, 1989.

[31] Y. W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Proceedings of the Game Developers Conference*, 2001.

[32] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[33] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[34] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, Microsoft Research, 2014.

[35] F. Bonchi, F. Gadducci, A. Kissinger, P. Sobociński, and F. Zanasi. String diagram rewrite theory I: Rewriting with Frobenius structure. *Journal of the ACM*, 69(2):14:1–14:58, 2022.

[36] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *Conference on Innovative Data Systems Research (CIDR)*, 2005.

[37] J. A. Bondy and U. S. R. Murty. *Graph Theory*, volume 244 of *Graduate Texts in Mathematics*. Springer, 2008.

[38] J. D. Brock and W. B. Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of Programming Concepts*, volume 107 of *LNCS*, pages 252–259, 1981.

[39] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*, pages 221–227, 2021.

[40] M. Brooker. Timeouts, retries, and backoff with jitter. Amazon Builders' Library, https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/, 2019. Accessed October 2026.

[41] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[42] K. Brown, E. Patterson, T. Hanks, and J. P. Fairbanks. Computational category-theoretic rewriting. In *Graph Transformation (ICGT 2022)*, volume 13349 of *LNCS*, pages 155–172, 2022.

[43] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[44] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, third edition, 2016.

[45] M. Buckland. *Programming Game AI by Example*. Wordware Publishing, 2005.

[46] 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*, 16(7):1601–1614, 2023.

[47] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14(1):70–93, 2016.

[48] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2nd edition, 2011.

[49] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *Proceedings of the 7th International Symposium on NASA Formal Methods (NFM)*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015.

[50] M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 263–274, 2013.

[51] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 38(4), 2015.

[52] Cash App. Redwood: Multiplatform reactive UI for Android, iOS, and web using Kotlin and Jetpack Compose. https://github.com/cashapp/redwood, 2024. Accessed October 2026.

[53] Cash App. Zipline: Run Kotlin/JS libraries in Kotlin/JVM and Kotlin/Native programs. https://github.com/cashapp/zipline, 2024. Accessed October 2026.

[54] J. Castillo. *Jetpack Compose Internals*. Leanpub, 2021.

[55] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2010.

[56] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[57] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538, 2005.

[58] S. Chaudhuri, K. Ellis, O. Polozov, R. Singh, A. Solar-Lezama, and Y. Yue. Neurosymbolic programming. *Foundations and Trends in Programming Languages*, 7(3):158–243, 2021.

[59] A. Cheung, N. Crooks, J. M. Hellerstein, and M. Milano. New directions in cloud programming. In *Conference on Innovative Data Systems Research (CIDR)*, 2021.

[60] P. Chiusano and R. Bjarnason. Unison: The big idea. https://www.unison-lang.org/docs/the-big-idea/, 2024. Accessed October 2026.

[61] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[62] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.

[63] ClickHouse. ClickHouse: an open-source column-oriented database for analytics. https://clickhouse.com/docs, 2026. Accessed October 2026.

[64] Cloudflare. Durable Objects: global uniqueness, input and output gates, SQLite storage, alarms and WebSocket hibernation. https://developers.cloudflare.com/durable-objects/, 2026. Accessed October 2026.

[65] Cloudflare. Durable Objects: data location and jurisdictions. https://developers.cloudflare.com/durable-objects/reference/data-location/, 2026. Accessed October 2026.

[66] Cloudflare. Cloudflare Queues: at-least-once delivery, retries and dead-letter queues. https://developers.cloudflare.com/queues/, 2026. Accessed October 2026.

[67] Cloudflare. Cloudflare Queues: limits. https://developers.cloudflare.com/queues/platform/limits/, 2026. Accessed October 2026.

[68] Cloudflare. Cloudflare Workers: isolates, service bindings, Hyperdrive and smart placement. https://developers.cloudflare.com/workers/, 2026. Accessed October 2026.

[69] Cloudflare. Cloudflare Workflows: durable steps, sleeps and external events. https://developers.cloudflare.com/workflows/, 2026. Accessed October 2026.

[70] Cloudflare. workerd: the JavaScript and Wasm runtime that powers Cloudflare Workers. https://github.com/cloudflare/workerd, 2026. Accessed October 2026.

[71] M. E. Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[72] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects (FMCO)*, volume 4709 of *LNCS*, pages 266–296, 2006.

[73] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, third edition, 2022.

[74] M. Costa, B. Köpf, A. Kolluri, A. Paverd, M. Russinovich, A. Salem, S. Tople, L. Wutschitz, and S. Zanella-Béguelin. Securing AI agents with information-flow control. arXiv preprint arXiv:2505.23643, 2025.

[75] P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Technical Report R.R. 88, Laboratoire IMAG, Université de Grenoble, 1977.

[76] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[77] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 315–326, 2007.

[78] U. Cubukcu, O. Erdogan, S. Pathak, S. Sannakkayala, and M. Slot. Citus: distributed PostgreSQL for data-intensive applications. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 2490–2502, 2021.

[79] Cyberspace Administration of China and six other agencies. Interim measures for the management of generative artificial intelligence services, 2023. Issued 10 July 2023; in force 15 August 2023.

[80] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 411–422, 2013.

[81] K. Dai, J. Hernando, Y. N. Billeh, S. L. Gratiy, J. Planas, A. P. Davison, S. Dura-Bernal, P. Gleeson, A. Devresse, B. K. Dichter, M. Gevaert, J. G. King, W. A. H. Van Geit, A. V. Povolotsky, E. Muller, J.-D. Courcol, and A. Arkhipov. The SONATA data format for efficient description of large-scale network models. *PLoS Computational Biology*, 16(2):e1007696, 2020.

[82] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.

[83] A. d'Avila Garcez and L. C. Lamb. Neurosymbolic AI: the 3rd wave. *Artificial Intelligence Review*, 56(11):12387–12406, 2023.

[84] A. P. Davison, D. Brüderle, J. M. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2:11, 2009.

[85] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *LNCS*, pages 625–635, 2021.

[86] J. Dean. Designs, lessons and advice from building large distributed systems. Keynote, ACM SIGOPS Workshop on Large-Scale Distributed Systems and Middleware (LADIS), 2009.

[87] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[88] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer, 1995.

[89] E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, and F. Tramèr. Defeating prompt injections by design. In *Proceedings of the IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pages 587–618, 2026.

[90] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[91] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[92] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 321–332, 2013.

[93] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, 21(2):437–446, 2022.

[94] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35(4):316–319, 2017.

[95] V. Dibia. *Designing Multi-Agent Systems*. 2025. Self-published.

[96] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[97] M. Djurfeldt, J. Hjorth, J. M. Eppler, N. Dudani, M. Helias, T. C. Potjans, U. S. Bhalla, M. Diesmann, J. Hellgren Kotaleski, and Ö. Ekeberg. Run-time interoperability between neuronal network simulators based on the MUSIC framework. *Neuroinformatics*, 8(1):43–60, 2010.

[98] E. Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Utrecht University, 2006.

[99] C. Dorn and H. Pribadi. TypeQL: a type-theoretic & polymorphic query language. *Proceedings of the ACM on Management of Data*, 2(2):1–27, 2024. PODS 2024.

[100] U. Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., 2007.

[101] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[102] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[103] G. T. Einevoll, A. Destexhe, M. Diesmann, S. Grün, V. Jirsa, M. de Kamps, M. Migliore, T. V. Ness, H. E. Plesser, and F. Schürmann. The scientific case for brain simulations. *Neuron*, 102(4):735–744, 2019.

[104] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[105] R. Elizarov, M. Belyaev, M. Akhin, and I. Usmanov. Kotlin coroutines: Design and implementation. In *Proceedings of the ACM SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 68–84, 2021.

[106] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, pages 263–273, 1997.

[107] Encore. Encore: infrastructure automation from application code. https://encore.dev, 2026. Accessed October 2026.

[108] N. C. Ene, M. Fernández, and B. Pinaud. Attributed hierarchical port graphs and applications. *Electronic Proceedings in Theoretical Computer Science*, 265:2–19, 2018.

[109] Epic Games. Unreal Engine 5.0 release notes. https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5.0-release-notes?application_version=5.0, 2022. Accessed October 2026.

[110] Epic Games. Blueprints visual scripting in Unreal Engine. https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprints-visual-scripting-in-unreal-engine, 2024. Accessed October 2026.

[111] Epic Games. Asset Registry (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/asset-registry-in-unreal-engine, 2026. Accessed October 2026.

[112] Epic Games. Balancing Blueprint and C++ (Unreal Engine 4.27 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/balancing-blueprint-and-cplusplus?application_version=4.27, 2026. Accessed October 2026.

[113] Epic Games. Blueprint compiler overview (Unreal Engine 5.8 documentation). https://dev.epicgames.com/

documentation/en-us/unreal-engine/compiler-overview-for-blueprints-visual-scripting-in-unreal-engine, 2026. Accessed October 2026.

[114] Epic Games. Blueprints visual scripting (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprints-visual-scripting-in-unreal-engine, 2026. Accessed October 2026.

[115] Epic Games. Exposing C++ to Blueprints visual scripting (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/exposing-cplusplus-to-blueprints-visual-scripting-in-unreal-engine, 2026. Accessed October 2026.

[116] Epic Games. Understanding the Gameplay Ability System (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/understanding-the-unreal-engine-gameplay-ability-system, 2026. Accessed October 2026.

[117] Epic Games. Unreal Insights (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-insights-in-unreal-engine, 2026. Accessed October 2026.

[118] Epic Games. Using Live Coding to recompile applications at runtime (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/using-live-coding-to-recompile-unreal-engine-applications-at-runtime, 2026. Accessed October 2026.

[119] Epic Games. MassEntity overview (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-mass-entity-in-unreal-engine, 2026. Accessed October 2026.

[120] Epic Games. Essential material concepts (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/essential-unreal-engine-material-concepts, 2026. Accessed October 2026.

[121] Epic Games. Nativizing Blueprints (Unreal Engine 4.27 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/nativizing-blueprints?application_version=4.27, 2026. Accessed October 2026.

[122] Epic Games. One File Per Actor (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/one-file-per-actor-in-unreal-engine, 2026. Accessed October 2026.

[123] Epic Games. Using Perforce as source control for Unreal Engine (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/using-perforce-as-source-control-for-unreal-engine, 2026. Accessed October 2026.

[124] Epic Games. Unreal Engine reflection system (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/reflection-system-in-unreal-engine, 2026. Accessed October 2026.

[125] Epic Games. Networking overview: replication and remote procedure calls (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/networking-overview-for-unreal-engine, 2026. Accessed October 2026.

[126] Epic Games. StateTree overview (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-state-tree-in-unreal-engine, 2026. Accessed October 2026.

[127] Epic Games. World Partition (Unreal Engine 5.8 documentation). https://dev.epicgames.com/documentation/en-us/unreal-engine/world-partition-in-unreal-engine, 2026. Accessed October 2026.

[128] Epic Games. Kismet visual scripting (Unreal Development Kit / Unreal Engine 3 documentation). https://docs.unrealengine.com/udk/Three/KismetHome.html, 2026. Accessed October 2026.

[129] Ericsson AB. OTP design principles: Release handling. https://www.erlang.org/doc/system/release_handling.html, 2024. Accessed October 2026.

[130] R. Fabian. *Data-Oriented Design*. Richard Fabian, 2018.

[131] M. Featonby. Making retries safe with idempotent APIs. Amazon Builders' Library, https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/, 2021. Accessed October 2026.

[132] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): an introduction. Technical Note CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006.

[133] M. Fernández, H. Kirchner, and B. Pinaud. Labelled port graph: A formal structure for models and computations. *Electronic Notes in Theoretical Computer Science*, 338:3–21, 2018.

[134] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[135] B. Fong. Decorated cospans. *Theory and Applications of Categories*, 30(33):1096–1120, 2015.

[136] B. Fong and D. I. Spivak. Hypergraph categories. *Journal of Pure and Applied Algebra*, 223(11):4746–4777, 2019.

[137] B. Fong and D. I. Spivak. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. Cambridge University Press, 2019.

[138] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

[139] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 1433–1445, 2018.

[140] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987.

[141] D. Garlan, R. T. Monroe, and D. Wile. Acme: an architecture description interchange language. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '97)*, pages 169–183, Toronto, Ontario, 1997.

[142] M.-O. Gewaltig and M. Diesmann. NEST (NEural Simulation Tool). *Scholarpedia*, 2(4):1430, 2007.

[143] S. Ghemawat, R. Grandl, S. Petrovic, M. Whittaker, P. Patel, I. Posva, and A. Vahdat. Towards modern development of cloud applications. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, pages 110–117, 2023.

[144] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[145] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2017.

[146] P. Gleeson, S. Crook, R. C. Cannon, M. L. Hines, G. O. Billings, M. Farinella, T. M. Morse, A. P. Davison, S. Ray, U. S. Bhalla, S. R. Barnes, Y. D. Dimitrova, and R. A. Silver. NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Computational Biology*, 6(6):e1000815, 2010.

[147] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 211–222, 2015.

[148] Google. FlexBuffers: schema-less binary encoding. https://flatbuffers.dev/flexbuffers/, 2024. Accessed October 2026.

[149] Google. Jetpack Compose: Thinking in Compose; the Compose runtime and Applier. https://developer.android.com/develop/ui/compose/mental-model, 2025. Accessed October 2026.

[150] Google. Dagger: A fully static, compile-time dependency injection framework for Java, Kotlin, and Android. https://dagger.dev/, 2026. Accessed October 2026.

[151] Google. Protocol Buffers documentation: Overview and language guide (proto3). https://protobuf.dev/, 2026. Accessed October 2026.

[152] Google. Enable app optimization with R8. https://developer.android.com/topic/performance/app-optimization/enable-app-optimization, 2026. Accessed October 2026.

[153] Google Cloud. Pub/Sub: ordering keys and exactly-once delivery. https://cloud.google.com/pubsub/docs, 2026. Accessed October 2026.

[154] D. K. Gosnell and M. Broecheler. *The Practitioner's Guide to Graph Data*. O'Reilly Media, 2020.

[155] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–210, 1989.

[156] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 31–40, 2007.

[157] J. Gregory. *Game Engine Architecture*. CRC Press, third edition, 2018.

[158] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing? Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*, pages 1–16, 2016.

[159] A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[160] P. R. Halmos. *Naive Set Theory*. Van Nostrand, 1960.

[161] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 156–166, 2014.

[162] N. Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22 (4):36–38, 1988.

[163] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[164] M. Hausenblas and S. Schimanski. *Programming Kubernetes*. O'Reilly Media, 2019.

[165] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, 2015.

[166] P. Helland. Life beyond distributed transactions: An apostate's opinion. In *Conference on Innovative Data Systems Research (CIDR)*, pages 132–141, 2007.

[167] J. M. Hellerstein and P. Alvaro. Keeping CALM: When distributed consistency is easy. *Communications of the ACM*, 63(9): 72–81, 2020.

[168] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6th edition, 2017.

[169] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245, 1973.

[170] M. Hicks and S. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6): 1049–1096, 2005.

[171] M. L. Hines and N. T. Carnevale. The NEURON simulation environment. *Neural Computation*, 9(6):1179–1209, 1997.

[172] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, 1992.

[173] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, 2016.

[174] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*, pages 150–155, 2017.

[175] B. Ibryam. Multi-runtime microservices architecture. InfoQ, https://www.infoq.com/articles/multi-runtime-microservice-architecture/, 2020. Published 27 February 2020. Accessed October 2026.

[176] ISO/IEC. ISO/IEC 39075:2024 Information technology – Database languages – GQL. https://www.iso.org/standard/76120.html, 2024. Accessed October 2026.

[177] J. Iyengar and M. Thomson. QUIC: A UDP-based multiplexed and secure transport. RFC 9000, Internet Engineering Task Force, 2021.

[178] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2012.

[179] D. Jemerov and S. Isakova. *Kotlin in Action*. Manning Publications, 2017.

[180] JetBrains. Kotlin/JS overview; kotlin symbol processing (KSP). https://kotlinlang.org/docs/js-overview.html, 2025. Accessed October 2026.

[181] JetBrains. Kotlin compiler plugins: K2 frontend (FIR) checkers and backend (IR) generation extensions. https://kotlinlang.org/docs/custom-compiler-plugins.html, 2026. Accessed October 2026.

[182] JetBrains. K2 compiler migration guide. https://kotlinlang.org/docs/k2-compiler-migration-guide.html, 2026. Accessed October 2026.

[183] JetBrains. Kotlin Multiplatform. https://kotlinlang.org/docs/multiplatform.html, 2026. Accessed October 2026.

[184] JetBrains. kotlinx.serialization: Kotlin multiplatform serialization. https://github.com/Kotlin/kotlinx.serialization, 2026. Accessed October 2026.

[185] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[186] T. Johnson, M. Amini, and X. D. Li. ThinLTO: Scalable and incremental LTO. In *Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–121, 2017.

[187] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36 (1):1–34, 2004.

[188] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall International, 1993.

[189] C. A. Joslyn and K. Nowak. Ubergraphs: A definition of a recursive hypergraph structure. *arXiv preprint arXiv:1704.05547*, 2017.

[190] K3s Project. K3s: Lightweight Kubernetes. https://k3s.io, 2026. Accessed October 2026.

[191] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proceedings of IFIP Congress 74*, pages 471–475. North-Holland, 1974.

[192] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[193] P. Katis, N. Sabadini, and R. F. C. Walters. Span(Graph): A categorical algebra of transition systems. In *Algebraic Methodology and Software Technology (AMAST)*, volume 1349 of *LNCS*, pages 307–321, 1997.

[194] J. Kepner et al. Mathematical foundations of the GraphBLAS. In *IEEE High Performance Extreme Computing Conference (HPEC)*, 2016.

[195] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 194–206, 1973.

[196] K. Kingsbury. Jepsen analyses. https://jepsen.io/analyses, 2013–2025. Accessed October 2026.

[197] M. Kivelä, A. Arenas, M. Barthelemy, J. P. Gleeson, Y. Moreno, and M. A. Porter. Multilayer networks. *Journal of Complex Networks*, 2(3):203–271, 2014.

[198] M. Kleppmann. Using logs to build a solid data infrastructure. In *Making Sense of Stream Processing: The Philosophy Behind Apache Kafka and Scalable Stream Data Platforms*, chapter 2, pages 39–79. O'Reilly Media, 2016.

[199] M. Kleppmann. How to do distributed locking. https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html, 2016. Accessed October 2026.

[200] M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, 2017.

[201] M. Kleppmann. Distributed Systems lecture series, University of Cambridge. https://www.youtube.com/playlist?list=PLeKd45zvjcDFUEv_ohr_HdUFe97RItdiB, 2020. Lecture videos. Accessed October 2026.

[202] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan. Local-first software: You own your data, in spite of the cloud. In *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 154–178, 2019.

[203] M. Kleppmann, D. P. Mulligan, V. B. F. Gomes, and A. R. Beresford. A highly-available move operation for replicated trees. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1711–1724, 2022.

[204] Klotho. Klotho: annotation-based infrastructure from code. https://klo.dev, 2026. No longer in active development. Accessed October 2026.

[205] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[206] J. Kodosky. LabVIEW. *Proceedings of the ACM on Programming Languages*, 4(HOPL):78:1–78:54, 2020.

[207] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[208] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *Proceedings of the NetDB Workshop*, 2011.

[209] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 611–626, 2023.

[210] S. Lack and P. Sobociński. Adhesive and quasiadhesive categories. *RAIRO – Theoretical Informatics and Applications*, 39(3): 511–545, 2005.

[211] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[212] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[213] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, 2001.

[214] L. Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[215] L. Lamport. Who builds a house without drawing blueprints? *Communications of the ACM*, 58(4):38–41, 2015.

[216] L. Lamport. The TLA+ Video Course. https://lamport.azurewebsites.net/video/videos.html, 2018. Lecture videos. Accessed October 2026.

[217] LangChain. LangGraph overview. https://docs.langchain.com/oss/python/langgraph/overview, 2026. Accessed October 2026.

[218] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[219] B. Leach. Designing robust and predictable APIs with idempotency. Stripe engineering blog, https://stripe.com/blog/idempotency, 2017. Accessed October 2026.

[220] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[221] T. Leinster. *Basic Category Theory*. Cambridge University Press, 2014.

[222] J. Leskovec. Stanford CS224W: Machine Learning with Graphs, Winter 2021. https://www.youtube.com/playlist?list=PLoROMvodv4rPLKxIpqhjhPgdQy7imNkDn, 2021. Lecture videos. Accessed October 2026.

[223] Q. Li, P. Kraft, C. Kozyrakis, M. Zaharia, and M. Stonebraker. DBOS: three years later. *The VLDB Journal*, 34(3), 2025. Article 37.

[224] Z. Li, J. Huang, and M. Naik. Scallop: a language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1463–1487, 2023.

[225] J. Linietsky. Godot 4.0 will discontinue VisualScript. Godot Engine blog, https://godotengine.org/article/godot-4-will-discontinue-visual-scripting/, 2022. Published 23 August 2022.

[226] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: a platform for secure distributed computation and storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 321–334, 2009.

[227] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[228] F. Lopez-Sancho Abraham. *Akka in Action*. Manning Publications, second edition, 2023.

[229] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 559–574, 2020.

[230] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[231] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[232] M. Madsen, M.-H. Yee, and O. Lhoták. From Datalog to Flix: A declarative language for fixed points on lattices. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 194–208, 2016.

[233] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC '95)*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer, 1995.

[234] R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt. DeepProbLog: neural probabilistic logic programming. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, pages 3749–3759, 2018.

[235] S. Marlow, L. Brandy, J. Coens, and J. Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 325–337, 2014.

[236] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[237] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[238] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1):330–339, 2010.

[239] Memgraph. Memgraph: A high-performance, in-memory graph database. https://memgraph.com/docs, 2026. Accessed October 2026.

[240] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology – CRYPTO '87*, volume 293 of *LNCS*, pages 369–378, 1987.

[241] Meta Platforms. Hermes: A JavaScript engine optimized for running React Native. https://github.com/facebook/hermes, 2024. Accessed October 2026.

[242] Meta Platforms. React: Preserving and resetting state; reconciliation. https://react.dev/learn/preserving-and-resetting-state, 2024. Accessed October 2026.

[243] S. Meyers. *Effective Modern C++*. O'Reilly Media, 2014.

[244] Microsoft. Debug Adapter Protocol specification. https://microsoft.github.io/debug-adapter-protocol/specification, 2026. Accessed October 2026.

[245] Microsoft. Language Server Protocol specification, version 3.18. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.18/specification/, 2026. Accessed October 2026.

[246] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[247] R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[248] S. Mimram and C. Di Giusto. A categorical theory of patches. *Electronic Notes in Theoretical Computer Science*, 298:283–307, 2013.

[249] J. C. Mogul. Emergent (mis)behavior vs. complex software systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '06)*, pages 293–304, 2006.

[250] M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

[251] A. Mokhov, N. Mitchell, and S. P. Jones. Build systems à la carte. *Proceedings of the ACM on Programming Languages*, 2 (ICFP):79:1–79:29, 2018.

[252] F. Mölder, K. P. Jablonski, B. Letcher, M. B. Hall, C. H. Tomkins-Tinch, V. Sochat, J. Forster, S. Lee, S. O. Twardziok, A. Kanitz, A. Wilm, M. Holtgrewe, S. Rahmann, S. Nahnsen, and J. Köster. Sustainable data analysis with Snakemake. *F1000Research*, 10:33, 2021.

[253] F. Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023.

[254] L. Moreau and P. Missier. PROV-DM: the PROV data model. https://www.w3.org/TR/2013/REC-prov-dm-20130430/, 2013. Editors. W3C Recommendation, 30 April 2013.

[255] L. Moreau and P. Missier. PROV-DM: The PROV data model. W3C Recommendation, 2013.

[256] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 561–577, 2018.

[257] R. Morris. MIT 6.824: Distributed Systems, Spring 2020. https://www.youtube.com/playlist?list=PLrw6a1wE39_tb2fErI4-WkMbsvGQk9_UB, 2020. Lecture videos. Accessed October 2026.

[258] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[259] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–142, 1997.

[260] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1 (1):97–123, 1990.

[261] N. Narkhede, G. Shapira, and T. Palino. *Kafka: The Definitive Guide*. O'Reilly Media, 2017.

[262] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 37–49, 2008.

[263] G. Neubig and D. Fried. CMU 11-768: AI Agents, Fall 2026. https://www.youtube.com/playlist?list=PLSN0qpDfUvTM, 2026. Lecture videos. Accessed October 2026.

[264] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[265] M. Newman. *Networks*. Oxford University Press, second edition, 2018.

[266] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[267] Nitric. Nitric: a multi-language framework with infrastructure from code. https://nitric.io/framework, 2026. Accessed October 2026.

[268] R. Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[269] R. Nystrom. *Crafting Interpreters*. Genever Benning, 2021.

[270] Object Management Group. Business Process Model and Notation (BPMN), version 2.0.2. OMG Document Number formal/2013-12-09, https://www.omg.org/spec/BPMN/2.0.2/, 2013. Accessed October 2026.

[271] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer. Hazelnut: A bidirectionally typed structure editor calculus. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 86–99, 2017.

[272] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

[273] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[274] OpenAPI Initiative. OpenAPI Specification, version 3.2.1. https://spec.openapis.org/oas/v3.2.1.html, 2026. Accessed October 2026.

[275] OpenTelemetry Authors. OpenTelemetry specification: context propagation and traces. https://opentelemetry.io/docs/specs/otel/, 2025. Accessed October 2026.

[276] S. Ouyang, W. Yu, K. Ma, Z. Xiao, Z. Zhang, M. Jia, J. Han, H. Zhang, and D. Yu. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

[277] R. Pang et al. Zanzibar: Google's consistent, global authorization system. In *USENIX Annual Technical Conference (ATC)*, pages 33–46, 2019.

[278] A. Parker, D. Spoonhower, J. Mace, R. Isaacs, and B. Sigelman. *Distributed Tracing in Practice*. O'Reilly Media, 2020.

[279] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[280] E. Patterson, O. Lynch, and J. Fairbanks. Categorical data structures for technical computing. *Compositionality*, 4(5), 2022.

[281] A. Pavlo. CMU 15-445/645: Intro to Database Systems, Fall 2024. https://www.youtube.com/playlist?list=PLSE8ODhjZXjYDBpQnSymaectKjxCy6BYq, 2024. Lecture videos. Accessed October 2026.

[282] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer. The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv:2302.06590, 2023.

[283] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, 1990.

[284] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek. Online, asynchronous schema change in F1. *Proceedings of the VLDB Endowment*, 6(11):1045–1056, 2013.

[285] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530, 2013.

[286] Reactive Streams. Reactive Streams specification for the JVM, version 1.0.0. https://www.reactive-streams.org/, 2015. Accessed October 2026.

[287] Reaktor. Reaktor data graph and Tidemark data layer. https://reaktor.build/docs/reaktor-data-graph, 2026.

[288] Reaktor. Reaktor: An extensible engine for applications and distributed systems – typed port hypergraphs, blueprint authoring, polyglot execution, and controlled evolution. Technical preprint, https://reaktor.build/docs/engine-preprint-graph, 2026. 16 September 2026.

[289] Reaktor. Reaktor FFI: Switchboard frames, cells, switches, trunks and wires. https://reaktor.build/docs/reaktor-ffi, 2026.

[290] Reaktor. Reaktor graph protocol (rgp/1). https://reaktor.build/docs/reaktor-graph-protocol, 2026.

[291] Reaktor. Reaktor graph vocabulary: node families N01–N40 and port profiles P01–P24. https://reaktor.build/docs/reaktor-graph-vocabulary-theory, 2026.

[292] Reaktor. Hyperedge theory: relation kinds H01–H36. https://reaktor.build/docs/reaktor-hyperedge-theory, 2026.

[293] Reaktor. Reaktor living graph: system architecture. https://reaktor.build/docs/reaktor-unified-system-graph, 2026.

[294] Reaktor. Reaktor graph language: One program across machines. Language design report, 2026. 7 October 2026.

[295] Reaktor. Reaktor stable foundations. https://reaktor.build/docs/reaktor-stable-foundations-5.6-ultra, 2026.

[296] T. Reps and T. Teitelbaum. The synthesizer generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48, 1984.

[297] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

[298] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[299] Restate. Restate: the durable runtime for agents and backends. https://restate.dev, 2026. Accessed October 2026.

[300] J. O. Ringert. Formal Methods for Software Engineering, Winter 2023/24, Bauhaus-Universität Weimar. https://www.youtube.com/playlist?list=PLGyeoukah9Nbd1yRDj3ridE7PtcL91-5u, 2023. Lecture videos. Accessed October 2026.

[301] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. R. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, P. Kohli, and A. Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625 (7995):468–475, 2024.

[302] M. Romero and B. Sewell. *Blueprints Visual Scripting for Unreal Engine 5*. Packt Publishing, third edition, 2022.

[303] A. Rotem-Gal-Oz. Fallacies of distributed computing explained. https://arnon.me/wp-content/uploads/Files/fallacies.pdf, 2006. White paper. Accessed October 2026.

[304] A. Sampson. CS 6120: Advanced Compilers: The Self-Guided Online Course, Cornell University. https://www.cs.cornell.edu/courses/cs6120/2020fa/self-guided/, 2020. Lecture videos. Accessed October 2026.

[305] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine. Survivable key compromise in software update systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 61–72, 2010.

[306] P. Sanz Leon, S. A. Knock, M. M. Woodman, L. Domide, J. Mersmann, A. R. McIntosh, and V. Jirsa. The Virtual Brain: a simulator of primate brain network dynamics. *Frontiers in Neuroinformatics*, 7:10, 2013.

[307] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[308] R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, and A. Milovidov. ClickHouse – Lightning fast analytics for everyone. *Proceedings of the VLDB Endowment*, 17(12):3731–3744, 2024.

[309] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, fourth edition, 2011.

[310] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *LNCS*, pages 386–400, 2011.

[311] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, 2010.

[312] G. Simhandl and U. Zdun. Cloud programming languages and infrastructure from code: an empirical study. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 143–156, 2024.

[313] A. Skiadopoulos, Q. Li, P. Kraft, K. Kaffes, D. Hong, S. Mathew, D. Bestor, M. Cafarella, V. Gadepally, G. Graefe, J. Kepner, C. Kozyrakis, T. Kraska, M. Stonebraker, L. Suresh, and M. Zaharia. DBOS: a DBMS-oriented operating system. *Proceedings of the VLDB Endowment*, 15(1):21–30, 2022.

[314] D. I. Spivak. Functorial data migration. *Information and Computation*, 217:31–51, 2012.

[315] D. I. Spivak. The operad of wiring diagrams: formalizing a graphical language for databases, recursion, and plug-and-play circuits. *arXiv preprint arXiv:1305.0297*, 2013.

[316] D. I. Spivak and B. Fong. MIT 18.S097: Applied Category Theory, January IAP 2019. https://ocw.mit.edu/courses/18-s097-applied-category-theory-january-iap-2019/, 2019. Lecture videos. Accessed October 2026.

[317] Standing Committee of the National People's Congress. Cybersecurity law of the People's Republic of China, 2016. Adopted 7 November 2016; in force 1 June 2017.

[318] Standing Committee of the National People's Congress. Data security law of the People's Republic of China, 2021. Adopted 10 June 2021; in force 1 September 2021.

[319] Standing Committee of the National People's Congress. Personal information protection law of the People's Republic of China, 2021. Adopted 20 August 2021; in force 1 November 2021.

[320] Stately. XState: state machines, statecharts and the actor model. https://stately.ai/docs/xstate, 2026. Accessed October 2026.

[321] M. Stimberg, R. Brette, and D. F. M. Goodman. Brian 2, an intuitive and efficient neural simulator. *eLife*, 8:e47314, 2019.

[322] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 553–564, 2005.

[323] Y. Sun and J. Han. Mining heterogeneous information networks: a structural analysis approach. *SIGKDD Explorations*, 14(2): 20–28, 2012.

[324] Supabase. Supabase: managed Postgres with row-level security. https://supabase.com/docs, 2026. Accessed October 2026.

[325] T. Sweeney. Welcome to Unreal Engine 4. Unreal Engine blog, https://www.unrealengine.com/en-US/blog/welcome-to-unreal-engine-4, 2014. Published 19 March 2014.

[326] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, pages 328–343, 2015.

[327] L. Tang, C. Bhandari, Y. Zhang, A. Karanika, S. Ji, I. Gupta, and T. Xu. Fail through the cracks: Cross-system interaction failures in modern cloud systems. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*, pages 433–451, 2023.

[328] V. Tankov, D. Valchuk, Y. Golubev, and T. Bryksin. Infrastructure in code: towards developer-friendly cloud applications. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1166–1170, 2021.

[329] Temporal Technologies. Temporal: Durable execution. https://docs.temporal.io, 2024. Accessed October 2026.

[330] Tencent. KuiklyUI: a cross-platform UI framework built on Kotlin Multiplatform. https://github.com/Tencent-TDS/KuiklyUI, 2026. Accessed October 2026.

[331] The Bazel Authors. Action graph query (aquery). https://bazel.build/query/aquery, 2026. Accessed October 2026.

[332] The Chromium Authors. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/, 2026. Accessed October 2026.

[333] The Dapr Authors. Dapr: Distributed application runtime. https://dapr.io, 2024. Accessed October 2026.

[334] The Git Project. Git wire protocol, version 2. https://git-scm.com/docs/gitprotocol-v2, 2026. Accessed October 2026.

[335] The Kubernetes Authors. Dynamic admission control (Kubernetes documentation). https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/, 2026. Accessed October 2026.

[336] The Kubernetes Authors. Controllers (Kubernetes documentation). https://kubernetes.io/docs/concepts/architecture/controller/, 2026. Accessed October 2026.

[337] The Kubernetes Authors. Custom resources (Kubernetes documentation). https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/, 2026. Accessed October 2026.

[338] The Kubernetes Authors. Operator pattern (Kubernetes documentation). https://kubernetes.io/docs/concepts/extend-kubernetes/operator/, 2026. Accessed October 2026.

[339] The Kubernetes Authors. Kubernetes API concepts: efficient detection of changes. https://kubernetes.io/docs/reference/using-api/api-concepts/, 2026. Accessed October 2026.

[340] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625 (7995):476–482, 2024.

[341] Unity Technologies. Entities package (Entities 6.7 manual). https://docs.unity3d.com/Packages/com.unity.entities@6.7/manual/index.html, 2026. Accessed October 2026.

[342] Unity Technologies. About Visual Scripting (Visual Scripting package 1.9 manual). https://docs.unity3d.com/Packages/com.unity.visualscripting@1.9/manual/index.html, 2026. Accessed October 2026.

[343] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[344] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[345] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 635–651, 2016.

[346] P. Veličković. Everything is connected: Graph neural networks. *Current Opinion in Structural Biology*, 79:102538, 2023.

[347] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.

[348] M. Vermeulen, R. Bjarnason, and P. Chiusano. *Functional Programming in Kotlin*. Manning Publications, 2021.

[349] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North-Holland, 1990.

[350] S. Weerawarana, C. Ekanayake, S. Perera, and F. Leymann. Bringing middleware to everyday programmers with Ballerina. In *Business Process Management (BPM 2018)*, volume 11080 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2018.

[351] P. Weisenburger, J. Wirth, and G. Salvaneschi. A survey of multitier programming. *ACM Computing Surveys*, 53(4):81:1–81:35, 2020.

[352] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger. Initialize once, start fast: Application initialization at build time. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):184:1–184:29, 2019.

[353] Winglang. Wing: a programming language for the cloud. https://winglang.io, 2026. Accessed October 2026.

[354] WSO2. Ballerina: a cloud-native programming language optimized for integration. https://ballerina.io, 2026. Accessed October 2026.

[355] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 662–676, 2017.

[356] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025.

[357] T. Xu and Y. Zhou. Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys*, 47(4):70:1–70:41, 2015.

[358] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*, pages 590–604, 2014.

[359] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, pages 50528–50652, 2024.

[360] Z. Yang, Z. Wu, M. Luo, W.-L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. S. Luan, G. Mittal, S. Shenker, and I. Stoica. SkyPilot: An intercloud broker for sky computing. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[361] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[362] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 249–265, 2014.

[363] N. Zeldovich. MIT 6.858: Computer Systems Security, Fall 2014. https://ocw.mit.edu/courses/6-858-computer-systems-security-fall-2014/, 2014. Lecture videos. Accessed October 2026.

[364] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–308, 2008.

[365] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan. Understanding and detecting software upgrade failures in distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, pages 116–131, 2021.

[366] J. Zhou et al. FoundationDB: A distributed unbundled transactional key value store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2653–2666, 2021.

A Proofs

Proposition 1. Set is adhesive, and adhesive categories are closed under functor categories and slices [210], so Set^C is adhesive for every small C. For a functor $T : C \to \mathsf{Set}$, the slice Set^C/T is equivalent to $\mathsf{Set}^{\int T}$, where $\int T$ is the category of elements of T, and is therefore again a topos. Adhesive categories are also closed under slices [210]. The local Church–Rosser, Parallelism and Concurrency theorems hold for DPO rewriting with monic rules in any adhesive category [102]. □

Proposition 2. For a functor $\iota : \mathcal{A} \to \mathcal{B}$ between small categories, $\Delta_\iota = - \circ \iota : \mathsf{Set}^{\mathcal{B}} \to \mathsf{Set}^{\mathcal{A}}$ has a left adjoint Σ_ι and a right adjoint Π_ι (left and right Kan extension) [314]. As a left adjoint it preserves colimits, in particular pushouts. As a right adjoint it preserves limits, in particular monomorphisms, since f is monic iff its kernel pair is trivial. A DPO derivation is a pair of pushout squares with monic rule legs, so its image is a pair of pushout squares with monic legs, i.e. a DPO derivation of the image rule. For typed graphs, apply the argument in $\mathsf{Set}^{\int T}$ with ι the inclusion $\int T_\ell \hookrightarrow \int T$. Well-formedness conditions that mention only sorts of layer ℓ are preserved, because Δ_ι restricts their interpretation; conditions that mention other layers are not part of the projected judgment. □

Theorem 3. (a) The encoding $\mathcal{E}$ sends ports to vertices, nodes to boxes labelled by kind and connections to boxes labelled by family, with vertex lists ordered by port and by role. It is injective on isomorphism classes because identities and roles are recorded in the boxes' arities. The correspondence between morphisms of the free hypergraph category on a monoidal signature and discrete cospans of hypergraphs over it is proved in [35]. The many-sorted form follows by replacing the single object with the set of port types. Composition is pushout along the shared boundary. It preserves (W4) when every boundary vertex is free of connection boxes on the side being mounted, which (W7) guarantees for scopes. (b) $\mathsf{Graph} = \mathsf{Set}^{\cdot \rightrightarrows \cdot}$ and its pointed variant are finitely complete. $\mathsf{Span}(\mathsf{C})$ is $\mathsf{Cospan}(\mathsf{C}^{\mathrm{op}})$, and cospans in a category with finite colimits form a hypergraph category [135, 136]. Hence $\mathsf{Span}(\mathsf{Graph})$ is a hypergraph category: spans compose by pullback, the tensor is the product, and the Frobenius structure on each object is given by diagonal spans. The free hypergraph category has the universal property that an assignment of objects and generators extends uniquely to a hypergraph functor [136]. Pullback over the interface alphabets synchronises transitions with equal labels, which is the composition of [193]. (c) Let a behaviour $B : J \to K$ be a span $A_J \leftarrow Q \to A_K$ of pointed graphs. Let $\mathrm{Tr}(B) \subseteq (A_J \times A_K)^*$ be the set of label sequences along paths from the initial vertex, with an idle label to model stuttering. A path in the pullback $Q_C \times_{A_K} Q_S$ is a pair of paths whose K-labels agree step by step. Hence $\mathrm{Tr}(C \circ S) = \{(j,l) \mid \exists k.\ (j,k) \in \mathrm{Tr}(S) \wedge (k,l) \in \mathrm{Tr}(C)\}$, taken stepwise. This expression is monotone in $\mathrm{Tr}(S)$, so $\mathrm{Tr}(S') \subseteq \mathrm{Tr}(S)$ implies $\mathrm{Tr}(C \circ S') \subseteq \mathrm{Tr}(C \circ S)$. Corollaries 4 and 5 follow by functoriality and by (c). □

Theorem 6. V^X is a complete lattice of height at most $h|X|$, and F, defined by (2), is monotone. By Knaster–Tarski it has a least fixpoint x^*, and since the height is finite, x^* is the limit of the Kleene chain from $\bot$. Consider any chaotic iteration that, starting from $\bot$, repeatedly replaces some coordinate x_v by $F_v(x)$. By induction, $x \le x^*$ throughout, since $x \le x^*$ implies $F_v(x) \le F_v(x^*) = x^*_v$. The iterates increase monotonically. Each strict increase raises one coordinate by at least one level, so there are at most $h|X|$ of them. Under fairness, every coordinate is re-evaluated after each change of its inputs, so the iteration stops at a fixpoint $\le x^*$, which by minimality equals x^*. A worklist re-evaluates exactly the elements reached through incidences of a changed element, giving $O(h|I|)$ message evaluations. Stale reads under asynchronous distribution are covered by [75]. □

Proposition 9. The preconditions, with $sub(n)$ the containment subtree of n:

Operation	Precondition (local)
add node n under q	fresh identifier; q exists; kind admissible under q's facets
remove node n	no children, ports or incidences remain (or the same patch removes them)
add port p to n	fresh identifier; the port kind is in the signature of $\kappa(n)$
add hyperedge e in scope s	fresh identifier; s exists
add incidence (e, x, r, k)	slot (e, r, k) free; upper bound of $\mu(r)$ respected; for a connection, $x \in P$ with kind $(\kappa(e), r)$ and no other connection; layer admissible; $x \in sub(sc(e))$ or a boundary port; contract compatible with the participants present
remove incidence	lower bound of $\mu(r)$ respected, or deferred to commit
move n under q	$q \notin sub(n)$; incident hyperedges remain inside their scopes
set attribute a of x	the current value equals the expected value; the new value is in the attribute's type

Each precondition is the local condition under which the corresponding judgment of Section 6.5 is maintained. The other judgments are unaffected because the operation does not touch the elements they constrain. Lower multiplicity bounds and global constraints (acyclicity of eager Bind, effect ceilings) are deferred: they are checked at commit over the patch's affected elements and declared read sets. □

Closure (3)**.** *Req* is monotone on the powerset lattice of elements, so C^* exists and is reached by a worklist in which each element enters at most once: $O(|X| + |I|)$. A region boundary that cuts a hyperedge of C^* is a candidate cut only if every cut incidence passes through a boundary port. Otherwise *Req* would have added the participant. □

Theorem 10. Parallel independence is the stated condition on matches; the Local Church–Rosser and Parallelism theorems for adhesive categories give the commuting derivations and the parallel rule [102, 210]. Identity-preserving rules mint the identifiers of created elements from the patch's identity and local names, independently of order, so both orders produce the same identifiers and hence equal graphs. □

B From the 2026 catalogue to Σ_{std}

The 2026 catalogue [291, 292] has 40 node kinds (N), 24 port profiles (P) and 36 hyperedge kinds (H). The tables give their images in Σ_{std}. The codes stay valid as names of derived forms, so existing documents keep their references. Angle brackets denote a profile, a facet or a derived kind (Table 5), and "tag" means a domain label with no semantics. Where a row offers two images, the script of Appendix C uses the first.

	Node kind	Σ_{std}		Node kind	Σ_{std}
N01	Pure computation	Component⟨pure⟩	N21	Database / state authority	Store
N02	Effectful operation	Component⟨effectful⟩	N22	Query / materialized view	Dataset⟨derived⟩
N03	Decision / pattern match	pure Component in a behaviour region	N23	Transaction / unit of work	Access at one Store
N04	Loop / recursive region	behaviour Scope with a back Transition	N24	Queue / log / pubsub	Channel (+ Resource realization)
N05	Actor / statechart	Component⟨exclusive⟩ + mailbox	N25	Cache / replica / sync	Replicates + replica Store
N06	Workflow / saga	Component⟨durable⟩ + Transition net	N26	Blob / buffer / asset store	Store⟨blob⟩
N07	Stream operator	Component⟨stream⟩	N27	Runtime host / executor	Place
N08	Fork / join / scheduler	Transition (join policy)	N28	Placement / resource pool	Place⟨pool⟩
N09	Timer / clock source	schedule Component on a place's clock + timer Channel	N29	Factory / dynamic instance	keyed Realizes (virtual activation)
N10	Function / blueprint region	Scope	N30	External system boundary	Scope⟨opaque⟩
N11	Surface / view component	Surface	N31	Schema / contract	Contract
N12	Route / navigation	Route	N32	Capability offer / req.	capability port; Contract⟨capability⟩
N13	Input / interaction source	Surface + Channel producer	N33	Policy / principal	Principal; policy as pure Component (split)
N14	Form / validation region	Surface + Scope	N34	Build / compiler pass	Derives; the pass's tool is a source-layer Artifact
N15	Device / platform adapter	Component⟨native⟩ offering capability ports	N35	Artifact / release manifest	Artifact
N16	Native resource / engine	Component⟨native⟩ or Resource	N36	Test / Shadow scenario	Component⟨check⟩
N17	Media transform	Component (tag)	N37	Claim / evidence record	Claim + Attests
N18	Model inference	Component⟨native⟩ (tag)	N38	Observer / query / visitor	a traversal; no node needed
N19	Tool / external API adapter	Component in an opaque Scope	N39	Agent / change proposer	Principal⟨agent⟩ + Component
N20	Human decision / approval	Component⟨human⟩	N40	Deployment / migration ctrl.	Component⟨controller⟩ + reconciler

Table 31: Node kinds.

	Port profile	Σ_{std} dimension		Port profile	Σ_{std} dimension
P01	Value / snapshot	BIND value or signal CHANNEL	P13	Completion / join token	TRANSITION join
P02	Request / response	SESSION call	P14	Cancellation / deadline	session control and runtime containment; not a port
P03	Durable command / receipt	SESSION call, durable acceptance	P15	Clock / tick / timer	CHANNEL from a clock capability
P04	Push event	event or topic CHANNEL	P16	Owned resource transfer	ownership move
P05	Pull iterator / cursor	SESSION stream (pull)	P17	Borrowed buffer / view	ownership borrow
P06	Credit-controlled stream	SESSION stream (credits)	P18	Shared read lease	ownership share
P07	Duplex session	SESSION duplex or multiparty	P19	Capability descriptor	capability BIND
P08	Signal / reactive state	signal CHANNEL	P20	Authorization request / grant	GRANT
P09	Mutation / conditional update	SESSION call + ACCESS(write)	P21	Lifecycle management	runtime containment; not a port
P10	Transaction participant	ACCESS with atomicity domain	P22	Checkpoint / migration	durable profile + DERIVES
P11	Execution / continuation	TRANSITION pre/post	P23	Artifact / schema reference	source-layer BIND or REALIZES
P12	Branch / guarded transition	TRANSITION guard	P24	Inspection / evidence	OBSERVES or ATTESTS; not a port

Table 32: Port profiles. P14, P21 and P24 become structure or assertions rather than ports. Ports of profiles P19, P20, P21, P23 and P24 that served only assertions or containment account for the 24% port reduction of Table 27.

	Hyperedge kind	Σ_{std}		Hyperedge kind	Σ_{std}
H01	Exclusive binding	BIND, exclusive selection	H19	Ownership / containment	containment forest
H02	Value flow	BIND, signal CHANNEL, stream SESSION (by law)	H20	Instantiation	keyed REALIZES
H03	Control precedence	TRANSITION	H21	Placement	PLACES
H04	Event subscription	fan-out CHANNEL	H22	Affinity / separation	PLACES near/apart
H05	Request / reply	SESSION call	H23	Scoped authorization	GRANT (grantor, grantee, resource, policy)
H06	Multiparty session	SESSION with a global type	H24	Capability resolution	capability BIND + support resolution
H07	Broadcast / fanout	fan-out CHANNEL	H25	Implements / source ownership	REALIZES (source to definition) + edit owner
H08	Merge-any	CHANNEL with several producers	H26	Artifact dependency	source-layer BIND
H09	Join-all	TRANSITION, join all	H27	Derivation / provenance	DERIVES
H10	Race / select	TRANSITION, join first	H28	Evidence supports claim	ATTESTS
H11	Keyed temporal join	stream Component over CHANNELS	H29	Activation cohort	release Artifact + REALIZES
H12	Barrier	TRANSITION, barrier	H30	Boundary export / alias	EXPORT
H13	Quorum requirement	TRANSITION quorum, or Store consensus profile	H31	Clock / time alignment	stream Component + clock contracts
H14	Replica group	REPLICATES	H32	Tensor / algebraic op.	behaviour inside a Component; not a relation
H15	Partition / route	PLACES, partitioned by key	H33	Fallback / preference set	BIND, ordered selection
H16	Atomic authority scope	ACCESS with atomicity at one Store	H34	Data access / lineage (retired)	ACCESS + GRANT + DERIVES
H17	Compensation	TRANSITION on failure (compensates)	H35	State migration	migration Artifact + DERIVES
H18	Supervision	runtime containment + supervision facet	H36	Navigation / interaction route	TRANSITION in navigation mode

Table 33: Hyperedge kinds.

C The bounded models and the corpus script

Both models are explicit-state, breadth-first explorations, so every counterexample reported is a shortest one. They are written as guarded commands. The artefact `anc/rg_models.py` implements them directly, and running `python3 anc/rg_models.py` prints Table 9.

Ownership handover. The state holds the authority's term t and owner w; each owner's view $view(o)$ (its term if it believes it owns, else 0); a send counter $sent(o)$; a multiset net of in-flight writes (o, t', k); and the set acc of accepted writes, each recorded with the term and owner at the moment of acceptance.

$$\begin{aligned}
\text{Send}(o) &: view(o) > 0 \wedge sent(o) < S \;\to\; net \mathrel{+}= (o, view(o), sent(o)),\; sent(o) \mathrel{+}= 1 \\
\text{Handover} &: h < H \;\to\; t \mathrel{+}= 1,\; w := next(w),\; view(w) := t,\; h \mathrel{+}= 1 \\
\text{Deliver}(m) &: m = (o, t', k) \in net \;\to\; net \mathrel{-}= m;\; \textbf{if } \neg fenced \vee t' = t \textbf{ then } acc \mathrel{+}= (o, t', t, w)
\end{aligned}$$

The invariant is $\forall (o, t', \tau, w') \in acc : t' = \tau \wedge o = w'$. The old owner is never told of the handover, which models a process that keeps running after it has lost ownership.

Effect identity. The state holds an attempt counter, multisets of in-flight requests and replies, the number of times the effect was applied, a deduplication table, the client's verdict (PENDING, APPLIED, NOT_APPLIED or UNKNOWN) and a duplication budget.

$$\begin{aligned}
\text{Send} &: \text{PENDING} \wedge a \le R \wedge (a = 0 \vee req = \emptyset) \;\to\; req \mathrel{+}= e,\; a \mathrel{+}= 1 \\
\text{Serve}(e) &: e \in req \;\to\; req \mathrel{-}= e,\; rep \mathrel{+}= \text{ok};\; \textbf{unless } dedup \wedge e \in table :\; applied \mathrel{+}= 1 \textbf{ and, if } dedup, table \mathrel{+}= e \\
\text{Drop, Dup} &: \text{lose any in-flight message; duplicate a request while the budget lasts} \\
\text{Reply} &: \text{ok} \in rep \wedge \text{PENDING} \;\to\; \text{APPLIED} \\
\text{GiveUp} &: a > R \wedge req = rep = \emptyset \wedge \text{PENDING} \;\to\; \text{NOT_APPLIED or UNKNOWN}
\end{aligned}$$

The invariant is $applied \le 1 \wedge \neg(\text{NOT_APPLIED} \wedge applied > 0)$.

The corpus script. The command

```
python3 anc/rg_catalog_eval.py anc/corpus out.json
```

reads the 140 definition graphs. `anc/corpus` is a snapshot of the corpus as published on 7 October 2026 at https://reaktor.build/docs/theory/. The script maps relation kinds, port profiles and node kinds to Σ_{std} with the tables of Appendix B, normalises roles and applies the checks of Section 20.1. It writes a summary and up to twelve examples per finding. The mapping tables are part of the script, so the result is reproducible and every judgment can be contested line by line.

C.1 TLA$^+$ templates checked with TLC

Five protocols that the design relies on are written as TLA$^+$ modules in `anc/`. Each is a template that the kernel would instantiate for a family or a release (Equation (1)):

- `EffectIdentity` for the Session call (Listing 25);
- `OwnershipHandover` for an exclusive component whose owner changes (Listing 26);
- `RegionRelease` for code push (Listing 6 in Section 9.5);
- `OutboxChannel` for a Channel fed by an outbox (Listing 27);
- `GraphWatch` for a watch of RGP (Listing 28).

Each module has a safe configuration and one configuration per design rule it tests, with that rule removed. TLC 2.19, from the TLA$^+$ tools 1.7.4, checks all sixteen configurations, and `make tlc` reruns them. Table 34 gives the results. Every safe configuration holds over its whole bounded state space, and every unsafe one fails with a short counterexample. The first two modules reach the same verdicts and counterexamples as the Python models of Table 9, and the effect-identity module has the same 87 states. The templates are hand-written today; generating them from the graph is the Q2 2027 work of Table 30.

Effect identity. Listing 25 transcribes the effect-identity model of Table 9. The variables are the client's attempt count, the requests and replies in flight (all copies of one effect), the number of times the effect was applied, whether the server has recorded the effect identity, the client's verdict, and the duplication budget. Without deduplication, TLC finds send, serve, retry, serve. When a timeout is read as "not applied", it finds a run that serves the request, loses the reply and both retries, and gives up with a false verdict.

```
---------------------------- MODULE EffectIdentity ----------------------------
EXTENDS Naturals

CONSTANTS MaxRetries, MaxDups, Dedup, TimeoutMeansUnknown
```

Module	Configuration	Invariant	Result	States	Steps
`EffectIdentity`	deduplication, timeout read as UNKNOWN	`AtMostOnce`, `NoFalseNegative`	holds	87	11
	no deduplication	`AtMostOnce`	violated		4
	timeout read as NOT_APPLIED	`NoFalseNegative`	violated		8
`OwnershipHandover`	the store fences by epoch	`OnlyCurrentOwnerWrites`	holds	326	11
	no fencing	`OnlyCurrentOwnerWrites`	violated		3
	sender checks its lease, no fencing	`OnlyCurrentOwnerWrites`	violated		3
`RegionRelease`	fence and drain	`NoWriteFromRetired`, `NoLostCallWrite`	holds	3,858	17
	no fence	`NoWriteFromRetired`	violated		6
	no drain	`NoLostCallWrite`	violated		4
`OutboxChannel`	atomic outbox, deduplication	`AtMostOnce`, `NoLostEffect`	holds	130	11
	effect and outbox written separately	`NoLostEffect`	violated		1
	no deduplication	`AtMostOnce`	violated		5
`GraphWatch`	starting cursor checked, epoch in the cursor, merging by addition	`Truthful`	holds	2,676	25
	starting cursor not checked	`Truthful`	violated		6
	cursor without the epoch	`Truthful`	violated		11
	merging keeps the latest delta	`Truthful`	violated		5

Table 34: TLC results. *States* counts the distinct states of a complete search. A search that finds a violation stops there, so only the length of the counterexample is given: *Steps* is the depth of a complete search, or the number of actions in the counterexample, which TLC's breadth-first search makes a shortest one.

```
VARIABLES attempts, req, rep, applied, seen, verdict, dups

vars == <<attempts, req, rep, applied, seen, verdict, dups>>

Init ==
    /\ attempts = 0
    /\ req = 0
    /\ rep = 0
    /\ applied = 0
    /\ seen = FALSE
    /\ verdict = "PENDING"
    /\ dups = 0

Send ==
    /\ verdict = "PENDING"
    /\ attempts <= MaxRetries
    /\ (attempts = 0 \/ req = 0)
    /\ req' = req + 1
    /\ attempts' = attempts + 1
    /\ UNCHANGED <<rep, applied, seen, verdict, dups>>

Serve ==
    /\ req > 0
    /\ req' = req - 1
    /\ rep' = rep + 1
    /\ IF Dedup /\ seen
          THEN UNCHANGED <<applied, seen>>
          ELSE (applied' = applied + 1 /\ seen' = TRUE)
    /\ UNCHANGED <<attempts, verdict, dups>>

DropRequest ==
    /\ req > 0
    /\ req' = req - 1
    /\ UNCHANGED <<attempts, rep, applied, seen, verdict, dups>>

DropReply ==
    /\ rep > 0
    /\ rep' = rep - 1
    /\ UNCHANGED <<attempts, req, applied, seen, verdict, dups>>

Duplicate ==
    /\ req > 0
    /\ dups < MaxDups
    /\ req' = req + 1
    /\ dups' = dups + 1
```

```
    /\ UNCHANGED <<attempts, rep, applied, seen, verdict>>

Reply ==
    /\ rep > 0
    /\ verdict = "PENDING"
    /\ rep' = rep - 1
    /\ verdict' = "APPLIED"
    /\ UNCHANGED <<attempts, req, applied, seen, dups>>

GiveUp ==
    /\ verdict = "PENDING"
    /\ attempts > MaxRetries
    /\ req = 0
    /\ rep = 0
    /\ verdict' = IF TimeoutMeansUnknown THEN "UNKNOWN" ELSE "NOT_APPLIED"
    /\ UNCHANGED <<attempts, req, rep, applied, seen, dups>>

Next == Send \/ Serve \/ DropRequest \/ DropReply \/ Duplicate \/ Reply \/ GiveUp

Spec == Init /\ [][Next]_vars

AtMostOnce == applied <= 1

NoFalseNegative == ~(verdict = "NOT_APPLIED" /\ applied > 0)

Safety == AtMostOnce /\ NoFalseNegative
================================================================================
```

Listing 25: The Session call with effect identity, in TLA$^+$.

Ownership handover. Listing 26 is the handover model of Table 9: the authority's term and owner, each owner's view of its own term, writes in flight, and the writes the store accepted, each with the term and owner at the moment of acceptance. The old owner is never told of a handover. Without fencing, TLC finds: the owner sends a write, ownership passes, and the delayed write is accepted. The same three steps defeat a sender that checks its lease before every write, because the check happens before the delay. Only the store, which sees the write when it commits, can refuse it. This is why `atomic(store)` checks epochs inside the store's transaction.

```
-------------------------- MODULE OwnershipHandover --------------------------
EXTENDS Naturals

CONSTANTS Owners, First, MaxSends, MaxHandovers, Fenced, SenderChecks

VARIABLES term, owner, view, sent, net, accepted, handovers

vars == <<term, owner, view, sent, net, accepted, handovers>>

Init ==
    /\ term = 1
    /\ owner = First
    /\ view = [o \in Owners |-> IF o = First THEN 1 ELSE 0]
    /\ sent = [o \in Owners |-> 0]
    /\ net = {}
    /\ accepted = {}
    /\ handovers = 0

Send(o) ==
    /\ view[o] > 0
    /\ sent[o] < MaxSends
    /\ SenderChecks => view[o] = term
    /\ net' = net \cup {[from |-> o, epoch |-> view[o], seq |-> sent[o]]}
    /\ sent' = [sent EXCEPT ![o] = @ + 1]
    /\ UNCHANGED <<term, owner, view, accepted, handovers>>

Handover ==
    /\ handovers < MaxHandovers
    /\ LET heir == CHOOSE p \in Owners : p # owner
       IN /\ owner' = heir
          /\ term' = term + 1
          /\ view' = [view EXCEPT ![heir] = term + 1]
    /\ handovers' = handovers + 1
    /\ UNCHANGED <<sent, net, accepted>>

Deliver(m) ==
    /\ net' = net \ {m}
```

```
    /\ IF Fenced => m.epoch = term
          THEN accepted' = accepted \cup
                 {[from |-> m.from, epoch |-> m.epoch, term |-> term, owner |-> owner]}
          ELSE UNCHANGED accepted
    /\ UNCHANGED <<term, owner, view, sent, handovers>>

Next ==
    \/ \E o \in Owners : Send(o)
    \/ Handover
    \/ \E m \in net : Deliver(m)

Spec == Init /\ [][Next]_vars

OnlyCurrentOwnerWrites ==
    \A a \in accepted : a.epoch = a.term /\ a.from = a.owner
=============================================================================
```

Listing 26: Ownership handover with fencing at the store, in TLA$^+$.

Outbox and channel. Listing 27 follows one producer, a relay and a consumer. The relay publishes each outbox record at least once and the broker may redeliver it. The consumer records each effect identity together with the effect it applies. `NoLostEffect` says that once nothing is pending, every committed effect has been applied once. Writing the effect and the outbox record in separate transactions loses an effect after a single step: the effect commits, and the record is never written. Without deduplication, one redelivery applies an effect twice.

```
--------------------------- MODULE OutboxChannel ---------------------------
EXTENDS Naturals

CONSTANTS Effects, MaxExtra, Atomic, Dedup

VARIABLES committed, outbox, channel, applied, seen, extra

vars == <<committed, outbox, channel, applied, seen, extra>>

Init ==
    /\ committed = {}
    /\ outbox = {}
    /\ channel = [e \in Effects |-> 0]
    /\ applied = [e \in Effects |-> 0]
    /\ seen = {}
    /\ extra = 0

Commit(e) ==
    /\ e \notin committed
    /\ committed' = committed \cup {e}
    /\ outbox' = outbox \cup {e}
    /\ UNCHANGED <<channel, applied, seen, extra>>

CommitWithoutOutbox(e) ==
    /\ ~Atomic
    /\ e \notin committed
    /\ committed' = committed \cup {e}
    /\ UNCHANGED <<outbox, channel, applied, seen, extra>>

Relay(e) ==
    /\ e \in outbox
    /\ channel' = [channel EXCEPT ![e] = @ + 1]
    /\ outbox' = outbox \ {e}
    /\ UNCHANGED <<committed, applied, seen, extra>>

RelayAgain(e) ==
    /\ e \in outbox
    /\ extra < MaxExtra
    /\ channel' = [channel EXCEPT ![e] = @ + 1]
    /\ extra' = extra + 1
    /\ UNCHANGED <<committed, outbox, applied, seen>>

Redeliver(e) ==
    /\ channel[e] > 0
    /\ extra < MaxExtra
    /\ channel' = [channel EXCEPT ![e] = @ + 1]
    /\ extra' = extra + 1
    /\ UNCHANGED <<committed, outbox, applied, seen>>
```

```
Consume(e) ==
    /\ channel[e] > 0
    /\ channel' = [channel EXCEPT ![e] = @ - 1]
    /\ IF Dedup /\ e \in seen
          THEN UNCHANGED <<applied, seen>>
          ELSE /\ applied' = [applied EXCEPT ![e] = @ + 1]
               /\ seen' = seen \cup {e}
    /\ UNCHANGED <<committed, outbox, extra>>

Next ==
    \E e \in Effects :
        \/ Commit(e) \/ CommitWithoutOutbox(e)
        \/ Relay(e) \/ RelayAgain(e) \/ Redeliver(e)
        \/ Consume(e)

Spec == Init /\ [][Next]_vars

Quiescent == outbox = {} /\ \A e \in Effects : channel[e] = 0

AtMostOnce == \A e \in Effects : applied[e] <= 1

NoLostEffect == Quiescent => \A e \in committed : applied[e] = 1
==============================================================================
```

Listing 27: A Channel fed by a transactional outbox, in TLA$^+$.

Watching a view. Listing 28 models one subscription of Section 12.4. The producer commits, sends single deltas or one merged delta while it has credit, and may drop deltas when it has none. The connection can be lost, and the producer can restart under a new epoch with an empty log. The subscriber applies a delta only if it starts at the subscriber's cursor, and otherwise asks for a reset. On reconnection the producer resumes from the subscriber's cursor or sends a reset. `Truthful` says that the subscriber's copy is the producer's state at some cursor of one epoch. Three configurations each remove one rule. Without the check of the starting cursor, TLC finds: two commits, a drop, a credit, a send, and a delivery that leaves a hole. Without the epoch in the cursor, the subscriber resumes after a restart at a position of the new epoch and mixes the two. When a merged delta keeps only the latest change, the first change is lost.

```
------------------------------ MODULE GraphWatch ------------------------------
EXTENDS Naturals, Sequences

CONSTANTS MaxCommits, MaxEpoch, MaxCredit, MaxDisconnects,
          Contiguous, EpochInCursor, ConflateBySum

VARIABLES epoch, head, sent, credit, net, connected, disconnects,
          cep, ccur, cstate, stale

link == <<net, connected, disconnects>>
client == <<cep, ccur, cstate, stale>>
vars == <<epoch, head, sent, credit, link, client>>

Prefix(ep, n) == {<<ep, j>> : j \in 1..n}

Delta(from, to, adds) ==
    [kind |-> "delta", ep |-> epoch, from |-> from, to |-> to, adds |-> adds]

Reset ==
    [kind |-> "reset", ep |-> epoch, from |-> 0, to |-> head, adds |-> Prefix(epoch, head)]

Init ==
    /\ epoch = 0 /\ head = 0 /\ sent = 0 /\ credit = 0
    /\ net = <<>> /\ connected = TRUE /\ disconnects = 0
    /\ cep = 0 /\ ccur = 0 /\ cstate = {} /\ stale = FALSE

Commit ==
    /\ head < MaxCommits
    /\ head' = head + 1
    /\ UNCHANGED <<epoch, sent, credit, link, client>>

Grant ==
    /\ connected /\ credit < MaxCredit
    /\ credit' = credit + 1
    /\ UNCHANGED <<epoch, head, sent, link, client>>

Send ==
    /\ connected /\ credit > 0 /\ sent < head
```

```
    /\ net' = Append(net, Delta(sent, sent + 1, {<<epoch, sent + 1>>}))
    /\ sent' = sent + 1
    /\ credit' = credit - 1
    /\ UNCHANGED <<epoch, head, connected, disconnects, client>>

SendMerged ==
    /\ connected /\ credit > 0 /\ sent + 1 < head
    /\ net' = Append(net, Delta(sent, head,
                IF ConflateBySum THEN {<<epoch, j>> : j \in (sent + 1)..head}
                                 ELSE {<<epoch, head>>}))
    /\ sent' = head
    /\ credit' = credit - 1
    /\ UNCHANGED <<epoch, head, connected, disconnects, client>>

Drop ==
    /\ connected /\ credit = 0 /\ sent < head
    /\ sent' = sent + 1
    /\ UNCHANGED <<epoch, head, credit, link, client>>

Accepts(m) ==
    /\ ~stale
    /\ Contiguous => m.from = ccur
    /\ EpochInCursor => m.ep = cep

Receive ==
    /\ net # <<>>
    /\ net' = Tail(net)
    /\ LET m == Head(net) IN
         IF m.kind = "reset" \/ Accepts(m)
            THEN /\ cstate' = IF m.kind = "reset" THEN m.adds ELSE cstate \cup m.adds
                 /\ ccur' = m.to
                 /\ cep' = m.ep
                 /\ stale' = FALSE
            ELSE /\ stale' = TRUE
                 /\ UNCHANGED <<cep, ccur, cstate>>
    /\ UNCHANGED <<epoch, head, sent, credit, connected, disconnects>>

Resync ==
    /\ connected /\ stale
    /\ \A i \in 1..Len(net) : net[i].kind # "reset"
    /\ net' = Append(net, Reset)
    /\ sent' = head
    /\ UNCHANGED <<epoch, head, credit, connected, disconnects, client>>

Disconnect ==
    /\ connected /\ disconnects < MaxDisconnects
    /\ connected' = FALSE /\ net' = <<>> /\ credit' = 0
    /\ disconnects' = disconnects + 1
    /\ UNCHANGED <<epoch, head, sent, client>>

Restart ==
    /\ epoch < MaxEpoch
    /\ epoch' = epoch + 1 /\ head' = 0 /\ sent' = 0 /\ credit' = 0
    /\ net' = <<>> /\ connected' = FALSE
    /\ UNCHANGED <<disconnects, client>>

Resume ==
    /\ ~connected
    /\ connected' = TRUE
    /\ IF ~stale /\ (EpochInCursor => cep = epoch) /\ ccur <= head
          THEN /\ sent' = ccur
               /\ net' = <<>>
          ELSE /\ sent' = head
               /\ net' = <<Reset>>
    /\ UNCHANGED <<epoch, head, credit, disconnects, client>>

Next ==
    \/ Commit \/ Grant \/ Send \/ SendMerged \/ Drop \/ Receive
    \/ Resync \/ Disconnect \/ Restart \/ Resume

Spec == Init /\ [][Next]_vars

Truthful == \E ep \in 0..MaxEpoch : cstate = Prefix(ep, ccur)
================================================================================
```

Listing 28: A watch of RGP, in TLA$^+$.

D Study plan

The paper draws on more fields than any one reader is likely to know. This appendix offers five reading paths, a map of the concepts, a table of what to read for each, and readings tied to the milestones of Table 30.

D.1 Reading paths

- **Practitioners and founders:** Sections 1, 2, 13, 17 and 18, then Section 22.
- **Theorists:** Sections 3 and 5 to 9 and Appendix A.
- **Implementers:** Sections 6, 8 to 11 and 13, Appendices C and E.5.
- **Reviewers of the evidence:** Section 20, Appendices B and C, and the scripts in `anc/`.
- **Builders of tools and agents:** Sections 4, 10.7, 16 and 19, then Section 20.4.

D.2 A map of the concepts

Figure 31 groups the background into four strands. Arrows point from a concept to the ones that build on it, and each strand ends in the sections it unlocks.

D.3 What to read, and how to check yourself

Table 35 lists, for each concept, where the paper uses it, a first source, a deeper source and a question to check understanding. Appendix E covers each concept briefly.

D.4 Readings by milestone

Table 36 pairs the quarters of the plan with the primary sources each needs. Each row ends in an artefact the milestone needs, so the reading is never separate from the build.

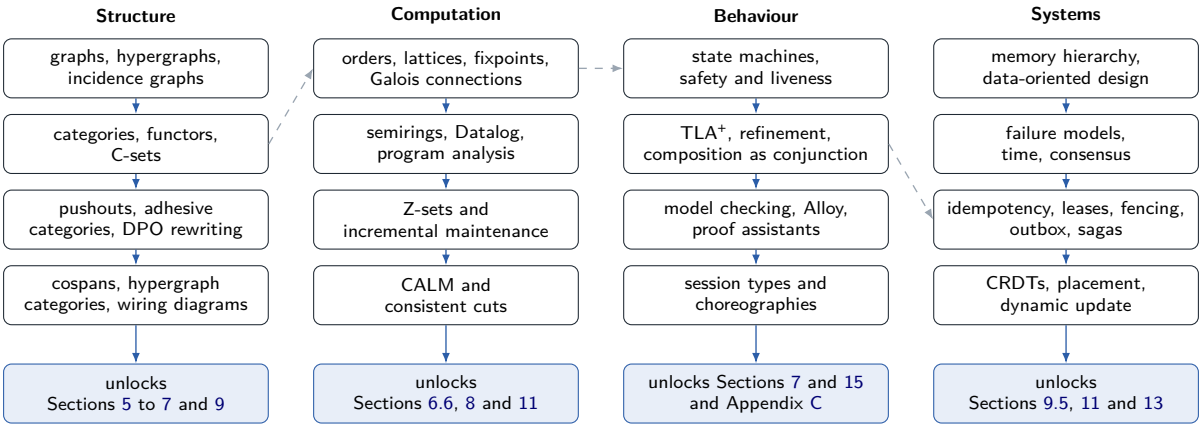


Figure 31: Four strands of background and the sections each one unlocks. Dashed arrows mark the main links between strands: an order is a category with at most one arrow between two objects; the reachable states of a state machine form a least fixpoint, which is what a model checker computes; and refinement states what idempotency and fencing must preserve.

Concept	Used in	Start with	Go deeper	Check yourself
Hypergraphs, incidence graphs	Section 5	[30], Appendix E.1	[37]	Why does a five-party session cost five incidences, and what does its clique of ten edges lose?
Network measures, multilayer networks	Section 21	[265]	[26, 197]	Which elements of Figure 1 would betweenness centrality flag as blast-radius risks?
Categories, C-sets	Section 5	[137], Appendix E.2	[221, 280]	Write the schema of directed graphs, one instance, and a morphism between two instances.
Pushouts, adhesivity, DPO	Section 9	[102]	[159, 210]	Why can a DPO rule not delete a node that still owns ports?
Cospans, hypergraph categories	Section 7	[137]	[35, 136, 193]	Compose two cospans by pushout; which ports become internal?
Lattices, fixpoints	Section 8	[82], Appendix E.3	[75, 266]	Model dependency resolution as a lattice. What do $\bot$ and $\top$ mean?
Semirings, Datalog	Section 8	[4]	[156, 232, 250]	Which semiring gives the worst-case latency of a call chain?
Z-sets, incrementality	Sections 8 and 11	[46]	[236, 251]	Why is deletion harder than insertion for incremental reachability?
CALM, consistent cuts	Section 8	[167]	[16, 56]	Why must “no provider exists” be evaluated on a consistent cut?
State machines, TLA$^+$, refinement	Sections 7 and 15	[214], Appendix E.4	[1, 2, 12]	Write the `Next` action of a lease handover with fencing.
Model checking, Alloy, proofs	Section 15	[24]	[85, 178]	What does a bounded check prove, and what does it not?
Session types, choreographies	Section 6	[173]	[50, 253]	Project a three-party protocol onto each of its roles.
Memory hierarchy, data-oriented design	Section 11	[100, 130], Appendix E.5	[5, 168]	How many cache lines does one scan of a column of 10^6 ports touch?
Failure models, consensus, leases	Sections 7 and 9.5	[48], Appendix E.6	[134, 230, 272]	Why can a client not tell a slow server from a crashed one?
Idempotency, outbox, sagas	Sections 9 and 13	[166]	[140, 200, 219]	Where must the deduplication record live for effects to happen exactly once?
CRDTs, local-first, placement	Section 17	[310]	[202, 360]	Which state can converge without coordination?
Dynamic software update	Section 9.5	[170]	[129, 207, 262]	What makes a code-push region closed?
Brain simulation, co-simulation	Section 17.1	[103]	[97, 146, 254]	Why does the smallest delay across a cut let two simulators advance independently?
Neurosymbolic programming	Section 17.1	[58]	[29, 224, 234]	Which semiring turns a proof rule into a probability, and which into a gradient?
Game engines, visual scripting	Section 4	[114, 124]	[113, 119, 125]	Which Blueprint nodes have no execution pins, and when do they run?
Dataflow and placement	Section 10.7	[191]	[3, 55, 220]	Why must an edge that crosses devices become a send and a receive?
Whole-program optimisation	Section 10.7	[88]	[186, 188, 352, 355]	Which optimisation becomes unsound if the other side of a boundary can be released on its own?
Failures in production	Section 16	[362]	[39, 174, 327]	How can retries at three layers keep a system down after its trigger is gone?
Coding agents	Section 19	[359]	[25, 227, 356]	What context does localisation need, and how would a traversal with coverage supply it?

Table 35: Concepts, where they are used, what to read and how to check yourself.

Quarter	Theory	Primary sources	Artefact
Q4 2026	C-sets, typed graphs, adhesive categories, DPO; information-flow lattices; failure models and fencing	[42, 91, 102, 199, 210, 280]	kernel v0; Alloy model of WF; principals and grants on the change path
Q1 2027	Attribute grammars, data-flow and semiring analysis, Datalog, incrementality, message passing; idempotency and outboxes; studies of production failures	[39, 46, 76, 166, 205, 232, 250, 362]	traversal engine; standard passes; channels with outbox; fault plans
Q2 2027	State machines, refinement, failure models, sessions, update; durable execution; whole-program and incremental compilation	[1, 53, 88, 170, 173, 186, 214, 253, 352]	RGL on four hosts; TLA$^+$ templates checked by TLC; Hermes profile; compilation plans
Q3 2027	Supervision, virtual actors, durable work, hypergraph categories, mechanised proof; placement; coding agents; visual composition	[20, 34, 35, 85, 114, 136, 193, 359, 360]	keyed actors; placement solver; Lean proofs; agent tools; graph-owned editor

Table 36: Readings by milestone.

E Primers

These primers give the background of Section 3 with worked examples, for readers who want it in one place. Appendix D says what to read next.

E.1 Hypergraphs and their incidence graphs

Take a session among a client c, a server s and an auditor a (Figure 3).

- *As a hyperedge* it is one element e with three incidences: $(e, c, \mathsf{client}, 0)$, $(e, s, \mathsf{server}, 0)$ and $(e, a, \mathsf{auditor}, 0)$.
- *As a graph* it would be a clique of three binary edges, c–s, s–a and a–c. The clique loses two facts: that the edges are one session, and which participant plays which role.
- *In general*, a hyperedge with k participants costs k incidences, while a clique costs $k(k-1)/2$ edges and loses the hyperedge's identity.

The *incidence graph* is bipartite. Its node sorts are elements $\{c, s, a\}$ and hyperedges $\{e\}$, with one edge per incidence. Kernels store this form, and graph algorithms run on it.

Network measures transfer directly:

- an element's degree is its number of incidences;
- betweenness centrality estimates how much dependency or traffic passes through an element, a proxy for blast radius;
- a sparse cut between communities is a candidate boundary for a region [265].

In a multilayer network each layer is a subgraph over a shared vertex set, with edges between layers [197]. In RG, connections stay within a layer, and assertions play the inter-layer role.

E.2 Categories in four steps

Categories and functors. A category has objects and arrows, an identity arrow on each object, and an associative composition of arrows. Sets and functions form the category Set. Any partial order is a category with at most one arrow between two objects. A functor maps objects to objects and arrows to arrows, preserving identities and composition [221].

C-sets. Directed graphs are the C-sets on the category $E \rightrightarrows V$, whose two arrows are s and t. An instance chooses sets and functions. For example, $V = \{1, 2, 3\}$ and $E = \{a, b\}$, with

$$s(a) = 1, \quad t(a) = 2, \quad s(b) = 2, \quad t(b) = 3.$$

A morphism of graphs is a pair of functions (f_E, f_V) that commutes with s and t, which is a natural transformation. The Reaktor schema (Figure 6) has more objects and arrows, but the principle is the same [280].

Gluing and rewriting. Let two scopes G and H share a boundary port p, given by monomorphisms $\{p\} \to G$ and $\{p\} \to H$. Their pushout $G +_{\{p\}} H$ is their disjoint union with the two copies of p identified.

A DPO rule that adds an interceptor to a call has three parts:

- L contains the session hyperedge e and the interceptor's port q;
- $K = L$;
- R is L plus the incidence $(e, q, \mathsf{interceptor}, 0)$.

Applying it deletes nothing and glues one incidence.

A rule that deletes a node n has $L = \{n\}$ and $K = R = \emptyset$. The *dangling condition* forbids applying it while n still owns ports. The rule must delete those ports and their incidences explicitly, which is why RG never leaves a dangling reference.

In an adhesive category, such as any category of C-sets, pushouts along monomorphisms behave like unions of sets [210]. That is what makes independent rules commute (Theorem 10).

Open systems. A cospan $J \to G \leftarrow K$ is G with an input boundary J and an output boundary K. Two cospans compose by pushout over the shared boundary (Figure 11). A *hypergraph category* adds the ability to split and merge wires, governed by the special Frobenius laws. This is exactly what a net does when it joins several ports [136]. Free hypergraph categories correspond to cospans of hypergraphs [35], which is the content of Theorem 3(a).

E.3 Fixpoints, semirings, Datalog and incrementality

Dependency resolution as a lattice. For each consumer port, take values in the flat lattice $\{\bot\} \cup \mathit{Providers} \cup \{\top\}$:

- $\bot$ means no candidate yet;
- $\top$ means ambiguous;
- joining two different providers gives $\top$.

A message along a Bind from provider p is p. Shadowing, where an inner scope hides the providers of outer ones, is not monotone, so it is computed first, in its own stratum: for each consumer, the nearest enclosing scope that offers the contract. The candidates in that scope are then joined. Within each stratum every function is monotone, so the least fixpoint exists and does not depend on order (Theorem 6). At the fixpoint each consumer holds one provider, $\bot$ (a missing-provider finding) or $\top$ (an ambiguity finding with both witnesses) [82, 266].

Path problems. With the Boolean semiring $(\vee, \wedge)$, path problems compute reachability, for example routes reachable from the root. With max-plus (max, +), they compute the worst-case latency of a call chain from per-hop budgets. With max-min, they compute bottleneck capacity [250].

Datalog. Reachability in Datalog is

$$\mathit{reach}(x, y) \leftarrow \mathit{edge}(x, y). \qquad \mathit{reach}(x, z) \leftarrow \mathit{reach}(x, y),\ \mathit{edge}(y, z).$$

Bottom-up evaluation computes the least model. "Unreachable" needs negation,

$$\mathit{unreachable}(r) \leftarrow \mathit{route}(r),\ \neg\, \mathit{reach}(\mathit{root}, r).$$

It is evaluated in a higher stratum, after *reach* is complete [4].

Incrementality. Inserting an edge (u, v) adds only pairs derived through it; semi-naive evaluation joins the delta, not the whole relation. Deletion is harder, because a pair may have other derivations. Z-sets (signed multisets) and DBSP's integration and differentiation operators handle both directions uniformly, at a cost proportional to the change for linear operators [46].

Coordination. Reachability is monotone: adding facts only adds conclusions, so the partitions of a graph can compute it without coordinating. "No path exists" is not monotone, and its answer is trustworthy only on a consistent cut [56, 167]. RG labels each traversal as one kind or the other (Proposition 8).

E.4 TLA$^+$ from zero

A TLA$^+$ specification declares variables, an initial predicate `Init` and a next-state action `Next`, and combines them as `Spec == Init /\ [][Next]_vars`.

- *Actions.* An action relates the current values of variables (unprimed) to their next values (primed). `UNCHANGED` lists the variables an action leaves alone.
- *Invariants.* An invariant is a state predicate. TLC evaluates it in every reachable state of a bounded instance and returns a shortest counterexample when it fails [214].
- *Liveness.* Properties of the form "eventually" need fairness conditions such as `WF_vars(Reply)`.

Reading Listing 25. Each action models one event:

- `Send`: the client sends or retries;
- `Serve`: the server applies the effect, or recognises its identity and does not apply it again;
- `DropRequest`, `DropReply` and `Duplicate`: the network misbehaves;
- `Reply`: the client learns the outcome;
- `GiveUp`: the client stops after its retries.

The counterexample without deduplication has four steps: send, serve, retry, serve. The other counterexample serves the request, loses the reply, loses two retries and gives up. If the client then concludes "not applied", it is wrong.

Refinement. An abstract specification, "the effect is applied at most once and reported truthfully", is refined by the protocol under a refinement mapping. Here the mapping sends the protocol's `applied` and `verdict` to the abstract ones [1].

What checking proves. Model checking proves a property for every behaviour within the bounds and nothing beyond them; proofs extend claims to all sizes [24, 85]. RG uses both: bounded checks for protocols, and proofs for the kernel's metatheory (Section 15).

E.5 Machines and data-oriented design

Table 37 gives orders of magnitude [86, 168]. Exact figures vary by hardware and region. The ratios are what matter for design.

Operation	Order of magnitude
Cache hit (first level)	~1 ns
Main memory access	~100 ns
Call across a native boundary (JNI, JSI, C ABI)	10 ns – 1 μs
Random read from solid-state storage	~100 μs
Round trip within a datacenter or region	~0.5–1 ms
Durable commit to a replicated store	~1–10 ms
Round trip across continents	~100 ms

Table 37: Orders of magnitude that shape placement and layout.

Layout. Consider scanning all ports of one family:

- *Array of structures.* Ports stored as objects, each with an owner, a family and a role, sit wherever the allocator put them. The scan touches every object header and every field.
- *Structure of arrays.* Stored as three integer arrays, the same scan reads one contiguous array of four-byte entries, sixteen per cache line, and lets the hardware prefetch [100, 130].

In compressed sparse row form, the incidences of hyperedge i are $ends[offsets[i] \,..\, offsets[i{+}1])$. For three hyperedges with two, three and one participants, $offsets = [0, 2, 5, 6]$.

A worked example. Count the cache lines that one scan of a million ports touches.

- *Objects.* On the JVM, a port object with a header, an owner reference, a family, a role, a contract reference and padding takes about 48 bytes, and the objects may be scattered. Reading every port's family touches up to a million cache lines of 64 bytes, and each miss to main memory costs about 100 ns (Table 37).
- *Columns.* The family column holds one byte per port, so the same scan reads 15,625 consecutive cache lines, about a megabyte, which the hardware prefetches.

The second scan touches 64 times fewer lines and no pointers. The same arithmetic explains the kernel's other choices (Section 11): hot columns apart from cold ones, integer codes instead of strings, pre-order intervals instead of parent walks, and one array instead of a map from ports to incidences.

Batches. Data-oriented code processes many rows per call. A traversal pass applies one message function to a whole CSR segment, and a patch is applied table by table, not element by element. Batches amortise dispatch, keep loops free of virtual calls, and let compilers vectorise. The same holds across runtimes: one FFI call that hands over a column slice costs about as much as one that hands over a single value.

Handles. A generational handle (slot, generation) detects reuse: freeing a slot increments its generation, so a stale handle no longer matches. In JavaScript engines, including Hermes, typed arrays give the same layout and avoid per-element objects and collection pauses.

Across runtimes. Table 37 explains why placement dominates. Co-locating two components that exchange many small messages turns milliseconds into nanoseconds. Batching amortizes per-call costs. Chattiness across a boundary that the code made invisible is a classic performance defect of distributed systems, which is why Bind never crosses a place.

E.6 Distributed systems, from failure to fencing

Failure model. Processes crash and restart. Messages are delayed, lost, duplicated or reordered, and clocks drift [48]. A timeout cannot tell a slow peer from a dead one, so after a timeout the effect of a call is unknown; Outcome has UNKNOWN for exactly this reason.

Idempotency. The caller chooses an effect identity. The authority that commits the effect records the identity in the same transaction, and answers a repeat with the stored outcome. Binding the identity to a digest of the request also catches a different request reusing the identity (Section 7) [166].

Leases and fencing. Suppose process A holds a lease with epoch 7 and pauses, for instance for garbage collection. The lease expires and B acquires epoch 8. When A wakes it writes with epoch 7. Only the store can stop that write, by rejecting any epoch below the highest it has seen [199] (Figure 4). This is the counterexample of Table 9, and the reason `atomic(store)` checks epochs inside the store's transaction.

Spanning services. Write the effect and an outbox record in one local transaction; a relay publishes the record after commit; consumers deduplicate by identity. A multi-step business process is a saga: each step has a compensation, and isolation, where it is needed, comes from reservations rather than locks [140].

Convergence without coordination. State whose merge is a join, such as grow-only sets, counters and last-writer-wins registers with logical timestamps, converges without coordination. These are CRDTs [310], and CALM says when an entire program can work this way.

Placement and data gravity. Put computation near the data it touches most, and keyed state near the users who share the key. A solver can weigh these against cost and capacity. Its constraints are the hard requirements of Section 6.6, and its output is a patch that admission checks.

F Glossary

Term	Meaning in this paper
Activation	A running instance of a definition element, with a key, an epoch, a lifecycle state and a parent activation (Section 6).
Admission	The predicate a host evaluates before it activates a release: trust, host profile, contracts, capabilities, grants, state plan, fresh evidence, policy (Section 9.5).
Assertion	A hyperedge that states a fact (placement, provenance, authority, evidence) and is never executed (Table 7).
Attributed C-set	A functor from a small category to Set with typed attribute values; the formal presentation of a Reaktor graph (Figure 6).
Boundary port	A port of a scope that is visible outside it, related to an inner port by Export; the leg of an open graph (Section 7).
Capability	Something a place can do, offered as a port and resolved against requirements by a support lattice. Not authority (Section 6.6).
Claim	An element that states a proposition about other elements, with evidence, a verifier and a status per dimension (Section 15).
Closure	The smallest set of elements that must change together with a region, a least fixpoint (Equation (3)).
Compilation plan	The facts a plan has fixed for one release region (bindings, placements, readers of each port, boundaries, profiles), pinned by digest and used by the compiler to specialise code (Section 10.7).
Connection	A hyperedge with operational semantics, of family Bind, Session, Channel or Transition.
Contract	A schema or protocol with a fingerprint and a compatibility relation. Ports match by contract.
Coverage claim	What a traversal or test saw and did not see: revision, layers, scopes, budget exclusions, opaque boundaries (Definition 8).
Cursor	A position in one producer's log under one epoch. A delta of a watch names the cursor it starts from and the cursor it ends at (Section 12.4).
Declassifier	A component holding a grant to lower an information-flow label for a stated purpose (Section 6.4).
Derived kind	A name for a base kind with required facets, ports and constraints, such as actor or workflow; a query, not a new kind (T3).
Differential observability	The defining trait of a gray failure: a component looks healthy to its own failure detector while its clients see it fail (Section 16).
Dual write	Writing two stores, or a store and a message system, without one transaction; a crash between the writes loses or invents an event. The outbox avoids it (Section 16).
Effect lattice	The ordered effect classes in which ceilings and approval thresholds are stated (Figure 10).
Elaboration	Running definition code over pinned inputs to produce a patch; deterministic and side-effect-free (Section 10).
Epoch	A number issued by an authority with each activation or lease; writes with a stale epoch are refused (Table 9).
Executor	A component that applies a plan under authority; the only place where traversals' results take effect (Section 8.4).
Export	A signed view of one authority's graph, restricted along a map of schemas and filtered by the importer's clearance, which another authority holds as a read-only replica (Section 12.8).
Facet	A typed attribute group that composes a node kind, as components compose entities in an entity–component system.
Family	A class of hyperedges with one set of roles and one set of laws; four connection and ten assertion families.
Fingerprint	A stable digest of a contract or plan; approval and binding refer to fingerprints.
Grant	An assertion of authority: a grantor allows a grantee to act on resources, with actions, expiry, delegation, rate and budget.

Term	Meaning in this paper
Hyperedge	A relation among any number of participants, each attached by an incidence.
Incidence	The attachment of one participant to one hyperedge, in a role and at a position.
Interceptor	A Session participant on the path between two roles: it receives each message first and passes, rejects or annotates it.
Label	An information-flow class on data and channels; flows may only rise in the label lattice.
Layer	One of source, definition, deployment, runtime, observation and assurance; a fibre of the graph.
Metastable failure	A state of overload that persists after its trigger is gone, sustained by extra work such as retries (Section 16).
Optimisation domain	The scope inside which the compiler may assume a closed world: the elements of a release region admitted together on one host at one generation, whose facts admission guarantees. Connections across hosts, generations or regions keep only their contracts (Section 10.7).
Outcome	The result of a session call: success, domain rejection, failure, or an unknown effect.
Patch	A typed double-pushout rule with a base revision; every change is one (Definition 9).
Place	Somewhere code runs or data lives: a device, isolate, process, runtime, host or region, with offered capabilities. Places nest.
Plan	A patch to the runtime or deployment layer computed by a traversal, with a fingerprint, applied by an executor.
Port	A typed interface owned by exactly one node: family, role, contract, ownership mode and quality of service.
Principal	A person, role, service account, agent or organisation that can hold grants.
Profile	A component's execution discipline: pure, effectful, exclusive, stream, durable, native or human.
Proposal	A patch with a base revision, author, checks and status, awaiting approval (Section 9).
Quality of service	The product of chains, per connection family, that a hyperedge requires and a binding guarantees (Table 6).
Realization	The binding that carries a connection: an in-process reference, an FFI trunk, RPC, a broker or a workflow engine.
Ref	A named pointer to a revision; branches and worktrees are refs.
Region	A scope selected for replacement as a unit; the subject of code push.
Retry amplification	The factor by which retries at several layers multiply the requests reaching the bottom layer; a max-times path problem over Session bindings (Listing 22).
Revision	An immutable root over versioned chunks of the graph's tables, with a digest.
Revision vector	One revision per consistency domain: the point at which a system is checked for truthfulness, and at which a read across authorities is evaluated (Definition 10 and Section 12.3).
RGP	The Reaktor Graph Protocol, `rgp/1`, through which authorities, hosts, tools, agents and peer authorities exchange graph facts and changes (Section 12).
Scope	The unit of composition, isolation, supervision and update; a node kind with boundary ports.
Signature	The vocabulary of kinds, families, roles and port kinds; Σ_{std} is the standard one.
Stack	A choice of realization for every capability class of the host profile, qualified per provider. The paper gives the reference stack and four others (Section 13.3).
Stub	A placeholder in a slice for an element outside it, giving only its identifier, kind and digest, or, for an element that a label withholds, only that it exists. Stubs keep a slice well formed (Section 12.3).
Support lattice	An ordered vocabulary of how well an offer meets a requirement (Definition 6).
Traversal	Typed message passing over incidences to a least fixpoint; returns a value, a patch or a plan, plus findings and coverage.
Trunk	A typed connection between two runtimes in one process, such as JSI between a Hermes engine and its host; it realizes Binds across runtime places and checks ownership modes.
Truthful system	One whose source, runtime, resources and claims agree with the graph, in the sense of Definition 10.
Typed hole	A placeholder for a required port or element in a draft; a draft with holes can be saved and analysed but not activated.
Version skew	Two or more versions of a component running at once during an upgrade, which must agree on every contract they share (Section 16).

G Roadmaps for learning and building

This appendix turns the paper into two roadmaps: one for learning what the design draws on, and one for building it. Both are drawn with Reaktor Graph itself, on facing pages (Figure 32, pages 112 and 113), and both are tabulated on the next pair of pages (Tables 39 and 40). Learning is always on the left page and building on the right. The roadmaps

carry no dates; the dated plan is Table 30. Here each step is a concept or a task, with what to read or deliver and what to practise or accept, and progress is measured by evidence rather than by the calendar.

The roadmaps as a graph. Each step is a Scope with an identifier.

- A learning step holds two Artifacts, what to read first and what to add, and a Claim: what the learner should be able to show.
- A building step holds the Artifact it delivers, the Observation that serves as evidence, and the Claim that the evidence must attest.
- Transitions give the suggested order of the steps within a track.
- Seven tracks face each other across the fold. Structure faces the kernel; computation, the traversals; behaviour, the assurance work; machines and data, the persistence layer; distributed systems, the families and their realizations; languages and compilers, RGL and its compiler; and engines, tools and agents, the tools and agents.
- A gate at the fold is one Transition whose join waits for its learning steps before its building steps begin. Gates show the needs within a track. Needs across tracks are listed in Table 40.

Why a graph rather than a list. The roadmap is a graph-owned scope like any other (Section 9), so the operations of the paper apply to it.

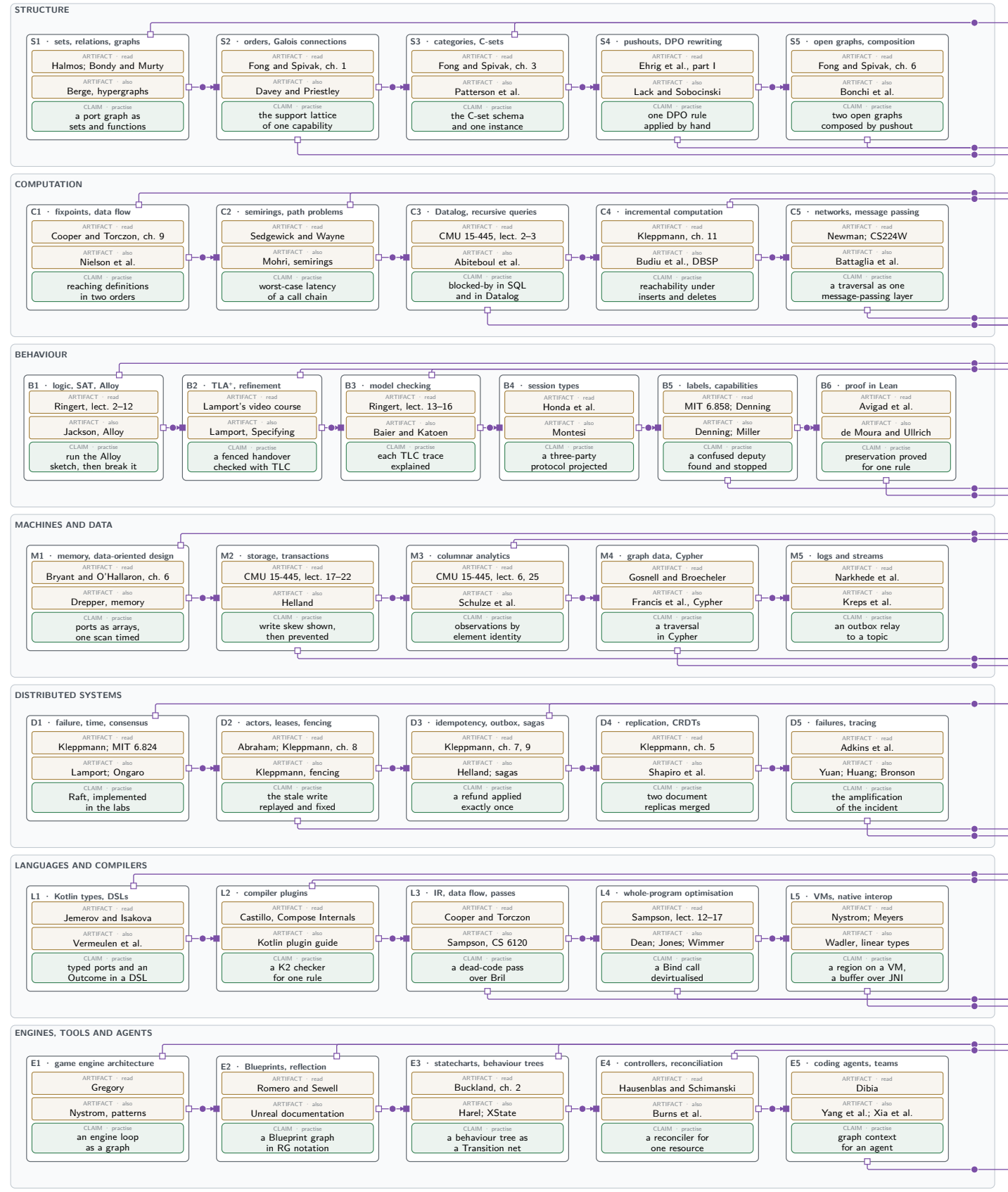
- *What can be started now?* The steps whose Transitions are enabled, because the claims of their predecessors are fresh. This is a traversal over the status of claims.
- *What blocks R3?* Reverse reachability from R3 over Transitions and gates, stopping at fresh claims.
- *What needs review?* The claims whose evidence has aged. A claim of mastery goes stale like any other claim (Section 15), so spaced review is the invalidation traversal applied to what a learner can show.
- *Who may sign off?* Practice that needs a reviewer is attested by a second principal. Rule A5 therefore applies to learning too: no one certifies their own mastery.

Changing the roadmap, for example to add a resource or to split a step, is a patch, reviewed as a graph diff.

Resources. Books and courses come from a curated catalogue of learning resources that the authors maintain. The catalogue links each resource to its chapters and to related resources, and the tables cite chapters and lectures from it. Papers are references of this paper. Where the catalogue has no resource, as for session types and for Lean, a step relies on papers and on an online book. Listing 29 shows how two learning steps, one building step and the gate between them are written in RGL.

```
val Learning = scope("learning", owner = Graph) {
    step("S1", "sets, relations, graphs") {
        read(book("Halmos", chapters = 6..10), book("Bondy and Murty", chapter = 1))
        practise(claim("a port graph as sets and functions"), fresh = 180.days)
    }
    step("S3", "categories, C-sets", after = "S2") {
        read(book("Fong and Spivak", chapter = 3), course("Applied Category Theory"))
        practise(claim("the C-set schema and one instance"), fresh = 180.days)
    }
}
val Building = scope("building", owner = Graph) {
    val tables = step("K1", "identity, C-set tables") {
        deliver(artifact("kernel v0 tables"))
        accept(claim("identifiers stable across runs"), evidence = observation("projection round trips"))
    }
    gate(join = All, from = Learning["S1", "S3"], to = tables)
}
```

Listing 29: Part of the roadmaps of Figure 32 in RGL (proposed API).



S3 · a step
ARTIFACT · read, or deliver
a book, a course, a deliverable
ARTIFACT or OBSERVATION
a paper to add, or the evidence
CLAIM · practise, or accept
what must be shown

Transition: the suggested order of the steps of one track

a gate at the fold: one Transition whose join waits for every learning step on the left before the building steps on the right begin

Every step is a scope. On the left page it holds what to read and what to practise; on the right page, what to deliver, the evidence that accepts it, and the claim that the evidence attests. A claim whose evidence ages goes stale and sends its step back to review. Tracks face each other across the fold, and identifiers index Tables 39 and 40.

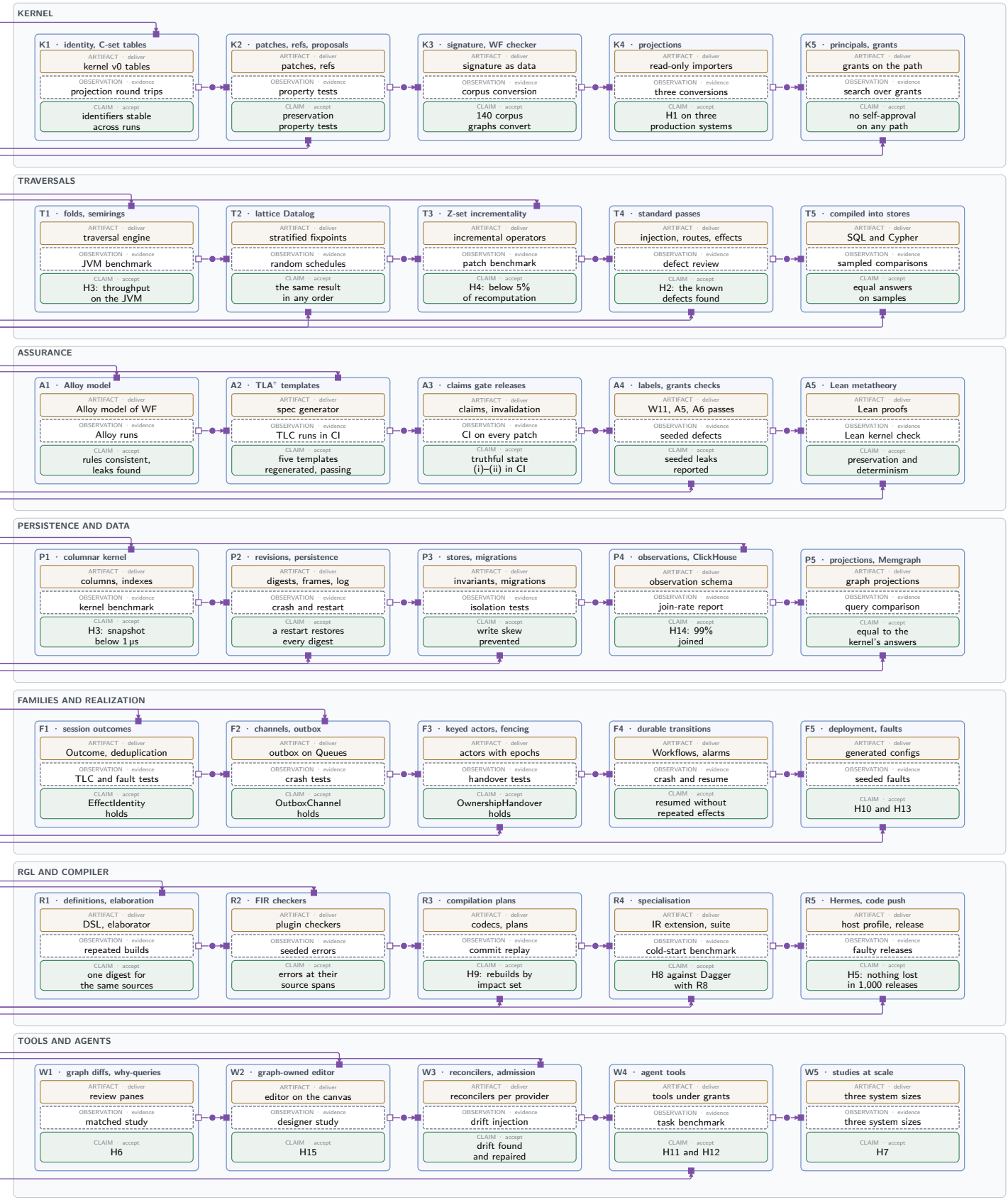


Figure 32: Roadmaps for learning (left page) and building (right page), drawn as one graph in RG notation. Seven tracks face each other across the fold. Within a track, Transitions give the suggested order; gates at the fold join the learning steps that a building step needs. Learning steps draw on a curated catalogue of books and courses and on the references of this paper; building steps end in the hypotheses of Section 20.4. There are no dates: progress is the evidence that attests each claim.

Step	Read	Practise	Unlocks
Structure			
S1 sets, relations, graphs	Halmos, sections 6–10 [160]; Bondy and Murty, ch. 1 [37]; also [30]	Write the chat scope of Figure 1 as sets of elements, ports and incidences with their maps; draw one session in the three forms of Figure 3.	K1
S2 orders, Galois connections	Fong and Spivak, ch. 1 [137]; also [82]	Derive the support lattice of one capability and the Galois connection between requirements and offers (Section 6.6).	K5, A4
S3 categories, C-sets	Fong and Spivak, ch. 3 [137]; their lectures [316]; also [221, 280]	Write the schema of Figure 6 as a category, one instance as a functor, and one morphism between two instances.	K1, K3
S4 pushouts, DPO rewriting	Ehrig et al., part I [102]; also [210]	Apply a patch to the chat scope by hand as a DPO rule, and show why deleting a node that still owns ports is refused.	K2
S5 open graphs, composition	Fong and Spivak, ch. 6 [137]; also [35, 136]	Compose two scopes along a boundary as cospans and list the ports that become internal (Corollary 4).	K2
Computation			
C1 fixpoints, data flow	Cooper and Torczon, ch. 9 [73]; Sampson's lectures on data flow [304]; also [75, 266]	Solve reaching definitions on a small program by chaotic iteration in two orders and check that the results agree (Theorem 6).	T1
C2 semirings, path problems	Sedgewick and Wayne, shortest paths [309]; also [194, 250]	Compute the max-plus latency and the max-times retry amplification of the incident of Section 16.5.	T1
C3 Datalog, recursive queries	Pavlo, lectures 2–3 [281]; also [4, 232]	Write the query of Listing 2 as recursive SQL and as Datalog, and explain why it terminates on cycles.	T2, T5
C4 incremental computation	Kleppmann, ch. 11 [200]; also [46, 236]	Maintain reachability over Z-sets under insertions and deletions, and explain why deletion is the hard case.	T3
C5 networks, message passing	Newman [265]; Leskovec's lectures [222]; also [29, 145]	Write Equation (2) as one message-passing layer, and rank the elements of Figure 2 by betweenness.	T4
Behaviour			
B1 logic, SAT, Alloy	Ringert, lectures 2–12 [300]; also [178]	Run Listing 17; find an instance, then a counterexample to a proposed change of the signature.	A1
B2 TLA$^+$, refinement	Lamport's video course [216]; also [1, 214]	Write the handover of Listing 26 from scratch and run TLC on its safe and unfenced configurations.	A2
B3 model checking	Ringert, lectures 13–16 [300]; also [24]	Explain each violated configuration of Table 34 from its trace.	A2, A3
B4 session types	also [173, 253]	Write a three-party checkout as a global type and project it onto each role.	F1
B5 labels, capabilities	Zeldovich et al. [363]; also [91, 162, 246, 259]	Find the confused deputy and the label leak that the graph of Figure 25d prevents, and the rules that catch them.	A4
B6 proof in Lean	Avigad et al. [22]; also [85]	State Listing 18 and prove it for the operation that adds an element.	A5
Machines and data			
M1 memory, data-oriented design	Bryant and O'Hallaron, ch. 6 [44]; Fabian, on tables and existence-based processing [130]; also [100]	Lay out the port table of Figure 18 as arrays and as objects, and time a scan of each.	P1
M2 storage, transactions	Pavlo, lectures 4–5 and 17–22 [281]; also [166]	Break a balance invariant by write skew under snapshot isolation, then prevent it with serializable isolation or one writer.	P2, P3
M3 columnar analytics	Pavlo, lectures 6 and 25 [281]; also [308, 322]	Load spans that carry element identifiers into a columnar store and answer a why-query by element.	P4
M4 graph data, Cypher	Gosnell and Broecheler [154]; also [17, 139]	Compile the traversal of Listing 2 into Cypher and compare its answer with the SQL one.	P5
M5 logs and streams	Narkhede et al. [261]; Kleppmann, ch. 11 [200]; also [198, 208]	Build an outbox relay to a topic and show that a crash between commit and publish loses nothing.	F2
Distributed systems			
D1 failure, time, consensus	Kleppmann's lectures [201]; MIT 6.824, lectures 1–7 [257]; also [211, 272]	Implement Raft in the 6.824 labs, and explain why a timeout cannot tell a slow peer from a dead one.	F1
D2 actors, leases, fencing	Abraham, the opening chapters on actors [228]; Kleppmann, ch. 8 [200]; also [34, 199]	Reproduce the stale write of Figure 4 and fix it with an epoch that the store checks.	F3
D3 idempotency, outbox, sagas	Kleppmann, ch. 7 and 9 [200]; also [131, 140, 166]	Make a refund call exactly once with an effect identity bound to the digest of its request.	F1, F2
D4 replication, CRDTs	Kleppmann, ch. 5 [200]; also [202, 310]	Merge two replicas of the shared document of Figure 25a and state its conflict law.	–
D5 failures, tracing	Adkins et al. [6]; Parker et al., on instrumentation [278]; also [39, 174, 362]	Compute the retry amplification of Section 16.5 and decide which single layer retries.	F5
Languages and compilers			
L1 Kotlin types, DSLs	Jemerov and Isakova, ch. 6, 9 and 11 [179]; Vermeulen et al., ch. 4 and 13 [348]; also [105]	Write the builder of Listing 7 with ports typed by family, role and contract, and an `Outcome` with four cases.	R1
L2 compiler plugins	Castillo, on the Compose compiler and runtime [54]; also [149, 181]	Write a K2 checker that reports W10 at the call site of an operation above its ceiling.	R2
L3 IR, data flow, passes	Cooper and Torczon, on intermediate representations and on optimisation [73]; Sampson, lectures 2–9 [304]	Write dead-code elimination and local value numbering over Bril.	R3
L4 whole-program optimisation	Sampson, lectures 12–17 [304]; also [88, 188, 352, 355]	Devirtualise a Bind call among the elements admitted together on one host, and state what would make it unsound.	R3, R4
L5 VMs, native interop	Nystrom, part III [269]; Meyers, on smart pointers and move semantics [243]; also [349]	Run a region's code on a bytecode VM, and pass a borrowed buffer across JNI without a copy or an escape.	R5
Engines, tools and agents			
E1 game engine architecture	Gregory, ch. 8 and 15–16 [157]; Nystrom, on game loops, components and data locality [268]	Draw an engine's loop and its systems in RG notation, marking what is native and what is composed.	W2
E2 Blueprints, reflection	Romero and Sewell, on the editor, functions, events and Blueprint communication [302]; also [113, 124]	Rebuild Figure 5a and translate it as in Figure 5b.	W2
E3 statecharts, behaviour trees	Buckland, ch. 2 [45]; Romero and Sewell, on behaviour trees [302]; also [163, 320]	Turn a behaviour tree into a Transition net with joins and guards.	W2
E4 controllers, reconciliation	Hausenblas and Schimanski [164]; also [47, 336]	Write a level-triggered reconciler for one custom resource and make it idempotent.	W3
E5 coding agents, teams	Dibia [95]; Fried and Neubig's lectures on agents [263]; also [227, 356, 359]	Give an agent a traversal with coverage instead of files, and count its round trips.	W4

Table 39: The learning roadmap, the left page of Figure 32: what to read for each step, how to practise it, and the building steps it unlocks. Books and courses come from a curated catalogue; papers are references of this paper. Chapter numbers of Kleppmann's book refer to its first edition.

Step	Needs	Deliver	Accept
Kernel			
K1 identity, C-set tables	S1, S3	C-set tables, stable identifiers and digests (Section 11)	projections round-trip with identifiers stable across runs
K2 patches, refs, proposals	S4, S5, K1	patches as DPO rules, refs and proposals (Section 9)	property tests of Proposition 9
K3 signature, WF checker	S3, K2	Σ_{std} as data with labels, tenancy and the effect lattice; $\mathsf{WF}_{\mathsf{static}}$	all 140 corpus graphs convert (Section 20.1)
K4 projections	K3	read-only projections of the current runtime's graphs, documents and deployment inventory	H1
K5 principals, grants	S2, K3	principals, grants and the effect lattice on the change path	no path lets a requester approve its own plan (A5)
Traversals			
T1 folds, semirings	C1, C2, K1	folds and semiring path problems over incidences (Equation (2))	H3: at least 10^7 incidences per second on the JVM
T2 lattice Datalog	C3, T1	lattice Datalog with strata	Theorem 6 under randomized schedules
T3 Z-set incrementality	C4, T2, K2	DBSP operators over revisions (Proposition 7)	H4
T4 standard passes	C5, T3	injection, routes, lifecycle, effects and flows, capability, closure and lints (Table 8)	H2
T5 compiled into stores	C3, T2	compilation into Postgres, Cypher and ClickHouse SQL (Listing 3)	answers equal to the kernel's on sampled graphs
Assurance			
A1 Alloy model	B1, K3	the schema and rules in Alloy (Listing 17)	an instance exists, and seeded inconsistencies are found
A2 TLA$^+$ templates	B2, B3, K3	templates instantiated per family and region (Equation (1))	the modules of Table 34 regenerate and pass TLC
A3 claims gate releases	B3, A2, T3	claims, evidence and invalidation (Section 15)	conditions (i)–(ii) of Definition 10 in CI
A4 labels, grants checks	S2, B5, K5	traversals for label flow, tenancy and authority	seeded label leaks, tenant leaks and self-approvals are reported
A5 Lean metatheory	B6, K2	proofs of Proposition 9 and Theorem 6	the proofs check in Lean 4
Persistence and data			
P1 columnar kernel	M1, K1	columns, CSR indexes, pre-order intervals and chunks (Section 11)	H3: snapshot below 1 µs and injection plan below 1 ms
P2 revisions, persistence	M2, P1	Merkle digests, the change log and FlexBuffers frames	a restart reproduces every digest
P3 stores, migrations	M2, K3	dataset invariants at commit, and migrations	seeded write skew is prevented
P4 observations, ClickHouse	M3, P2	observations joined by element identity in ClickHouse	H14
P5 projections, Memgraph	M4, T5	projections of the system and data graphs in Memgraph	Cypher answers equal the kernel's
Families and realization			
F1 session outcomes	B4, D1, D3, K3	Outcome with UNKNOWN, effect identity and deduplication	the EffectIdentity template holds for the realization
F2 channels, outbox	D3, M5, F1	`atomic(store)` with an outbox on Cloudflare Queues	OutboxChannel holds, and crash tests lose nothing
F3 keyed actors, fencing	D2, F1	keyed actors on Durable Objects and JVM shards, with epochs	OwnershipHandover holds
F4 durable transitions	F2	durable transitions on Workflows and Durable Object alarms; schedules	runs resume after crashes without repeating effects
F5 deployment, faults	D5, F4	the deployment layer for Cloudflare, Google Cloud and Postgres; fault plans	H10 and H13
RGL and compiler			
R1 definitions, elaboration	L1, K2	definitions and their elaboration into patches (Section 10)	the same sources give the same digest
R2 FIR checkers	L2, R1	checkers for W5 and W9–W11	seeded errors are reported at their spans
R3 compilation plans	L3, L4, T4, R2	codecs and compilation plans (Listing 10)	H9
R4 specialisation	L4, R3	the IR extension and the conformance suite	H8
R5 Hermes, code push	L5, R4, F5	the Hermes host profile and region release (Section 9.5)	H5
Tools and agents			
W1 graph diffs, why-queries	K2, T4	graph diffs ranked by risk, with why-queries	H6
W2 graph-owned editor	E1, E2, E3, W1	editing graph-owned scopes on the canvas, as patches	H15
W3 reconcilers, admission	E4, F5	reconcilers and admission per host	condition (iii) of Definition 10
W4 agent tools	E5, K5, A4	query, propose, check, plan and observe as tools under grants	H11 and H12
W5 studies at scale	W1, W4	studies at 10^2, 10^3 and 10^4 elements	H7

Table 40: The building roadmap, the right page of Figure 32: what each step needs, from both pages, what it delivers, and what accepts it. Acceptance names a hypothesis of Section 20.4 where one applies.