The Work Graph

An engine for projects carried out by people, agents and experts

Shibasis
BestBuds Ventures

12 November 2026, revised 16 November 2026 · Design paper

Abstract

Projects in every discipline are now carried out by mixed groups of people, AI agents, contractors and certified experts, often across several organisations, and sometimes by one person directing many agents. Every such group needs four answers at any moment: what it is trying to achieve, who owes what to whom, what evidence changed the plan, and who can decide next. Empirical studies show that coordination consumes much of the working day, few large projects meet their plans, work flows when promises are kept reliably, shared models catch conflicts before construction, and people teamed with AI gain only when each takes the tasks it does best. This paper proposes the *work graph*, a shared engine for multi-actor projects. Four kinds of element and six families of reified assertion record outcomes, commitments, evidence and decisions, turning the four questions into queries. Its unit is the *commitment*: a promise, agreed between a debtor and a creditor, either to achieve a proposition by a deadline or to maintain an invariant, whose acceptance query belongs to the creditor. Ten verbs govern commitments. An optimistic admission gate separates a debtor's exclusive *write lease* over its declared region from a read-tracked *semantic guard* over everything its acceptance query and antecedent inspect, and it returns a witness with every refusal. The work graph runs over any *subject space*, from a folder of files to a building model, and it is the coordination part of an extended port-hypergraph meta-model, so that coordination and structured system models can share one concurrent persistent hypergraph store. We prove that two admissible changes of which neither sets a predicate the other reads commute, and that the admitted history is serialisable. We implement the whole engine in Kotlin and its concurrent graph store in C++23, with wait-free snapshot pinning, a short serialised commit gate and an off-graph liveness wheel, and reproduce both, with their tests, in full. We trace five cross-disciplinary projects through the engine and state falsifiable hypotheses.

Keywords: coordination; commitments; multi-actor projects; human–agent teams; future of work; markets for work; optimistic concurrency; multiversion concurrency control; port hypergraphs.

Status of claims. This paper specifies an engine and reports no new empirical measurements of its own; empirical claims rest on published studies and named industry surveys. Portions of the state described here already run under other names in two products: Hangar, where Reaktor's agents build software [91], and Manna, where people organise projects, learning paths and resources [77]. As of 16 November 2026 neither product uses the engine of this paper. A companion document reports the software release measurements that motivated this design and its build order [93]. The engine is implemented twice and reproduced in full in the appendices: a C++23 concurrent graph store, whose 40 checks pass in an optimised build and under the address, undefined-behaviour and thread sanitizers (Appendix D, run on 16 November 2026), and a Kotlin engine for the whole specification, whose 59 tests passed on 12 November 2026 (Appendix E). The tests show that the design behaves as specified on the scenarios they cover. Neither implementation is deployed in a product, and neither has been timed. Theorem 1, Proposition 2, and Corollary 3 are proved in Appendix A.

Contents

1 Introduction 2
2 The problems of multi-actor work 3
3 Evidence from research, experiments and industry 4
4 Division of labour: people, agents and experts 6
5 The work graph 6
6 Commitments and verbs 9
7 Admission against commitments 10
8 Liveness and attention 12
9 Modes, matching and markets 13
10 Subject spaces with structure 14
11 Five projects through the engine 15
12 How the engine helps 17
13 One graph model and concurrent engine 18
14 The engine in Hangar and Manna 22
15 Hypotheses and evaluation 23
16 Risks 23
17 Related work 24
18 Conclusion 25
A Proofs 29
B Existing tools and the work graph 30
C Glossary 31

D C++23 concurrent graph 33 E Kotlin reference engine 50

1 Introduction

Most work worth doing is now carried out by groups whose members are of different kinds. A summer-school research project at Neuromatch Academy brings together three students in different countries, a mentor, a teaching assistant, and agents that search the literature and run analyses [115]. A wedding joins a couple, two families, a planner, a venue, a caterer, a photographer, and agents that compare quotes and maintain the guest list. A house is built by an owner, an architect, a structural engineer whose stamp the permit requires, a general contractor with its subcontractors, an inspector and a lender. A software startup may consist of three founders directing dozens of coding and verification agents. One maker can now take on a project that once needed a studio, such as an anatomical simulation of a brain in a game engine, by directing agents and buying a few hours of specialist review.

Every one of these groups needs four answers at any moment: what the group is trying to achieve, who owes what to whom, what evidence changed the plan, and who can decide next. In practice those answers sit scattered across people's heads, chat threads, email chains, contracts and issue trackers, so someone has to ask for them. Keeping one another informed takes roughly half the working day, yet large projects still routinely miss their schedules and budgets (Section 3). AI agents make each of these failures worse. An agent that cannot see who owns an area either stalls or overwrites another actor's work; an agent allowed to grade its own work often reports completion it did not achieve; and a person who cannot see what agents are waiting on spends the day asking them.

This paper proposes the *work graph*, a coordination engine shared by every actor in a project, whether a person, an agent, an organisation or a licensed expert. Four kinds of element and six families of assertion record outcomes, commitments, evidence and decisions, turning the four questions into graph queries (Section 5). Its unit is the *commitment* from multi-agent systems [101, 125]: a debtor promises a creditor, with bilateral consent, either to achieve a proposition by a deadline or to maintain an invariant across time, while the creditor retains the test of acceptance (Section 6). Ten verbs evolve commitments, and every change passes through a single admission gate that protects active promises and returns an actionable witness whenever a change is refused (Section 7). Rules for liveness, review windows and attention ranked by cost of delay turn the measured bottlenecks of group work into rules of the protocol (Section 8). Six collaboration modes, routing by comparative advantage and markets for complex work are configurations of the same engine (Section 9).

An engine for any discipline. The work graph governs coordination and leaves the storage of the domain to others. The artifacts being produced remain in their *subject space*, such as a shared drive, a code repository, a building information model (BIM), a versioned dataset or a booking calendar. The work graph refers to them without owning them, and it becomes more precise as that space gains structure (Section 10). Over flat documents, a region names a file and people declare what each check reads. Over a 3D building model, a region selects a single beam or duct, and spatial clashes are caught at commit time before anything is fabricated. When the subject space is itself a typed graph model, an extended port-hypergraph meta-model holds coordination and subject structure in one persistent concurrent hypergraph store (Section 13).

Each actor at what it does best. Controlled trials show that human–AI pairs excel inside a model's competence yet degrade outside it, outperforming individuals only when tasks divide along comparative advantage (Section 3). The work graph holds every actor to the same discipline: act under bounded grants, make explicit commitments, attach evidence and wait for the creditor's acceptance. Tasks can therefore go to an agent for broad search, to a person for ambiguous judgment, or to a licensed professional for statutory sign-off, with little lost at each hand-off (Sections 4 and 9.2). Posted as an open commitment backed by escrowed resources and trust-weighted evidence, the same primitive forms a market for complex work (Section 9.3).

Contributions.

- **C1 Problems, evidence and division of labour** (Sections 2 to 4). Fifteen recurring coordination failures grounded in empirical studies across disciplines, distilled into eight engine requirements (**R1**–**R8**).
- **C2 The work graph** (Section 5). Four element kinds and six addressable assertion families over a joint predicate space, supporting grants with revocation, verifiable credentials, trust-weighted capabilities and the four status queries.
- **C3 Commitments and verbs** (Section 6). Achievement and standing commitments with bilateral consent, creditor review windows, ten lifecycle verbs, and a split between a commitment's declared lease region R_0 and its read-tracked guard region R_{obs}.
- **C4 Optimistic admission against commitments** (Section 7 and Appendix A). Five admission clauses separating exclusive write leases from non-regression guards and additive resource escrow, explicit refusal witnesses, and proofs that two admissible changes of which neither sets a predicate the other reads commute and that the admitted history is serialisable (Theorem 1 and Proposition 2).
- **C5 Liveness, modes, matching and markets** (Sections 8 and 9). Heartbeats, recorded waits, acyclic dependency synthesis, root-cause-coalesced attention ranking, six collaboration modes, comparative-advantage matching and escrow-backed markets.

C6 Structured subject spaces and migration (Section 10). Exact regions, store-recorded read sets, stale-evidence propagation, invariant preconditions and a five-step shadow migration protocol.

C7 One unified graph model and two implementations (Section 13 and Appendices D and E). An extended port-hypergraph schema, stated as a category of presheaves, that holds the work graph and typed system models together; a C++23 concurrent graph store with epoch-pinned snapshots, recorded read sets, a short serialised commit gate and an off-graph timing wheel; and a Kotlin engine implementing the whole specification. Both are tested and reproduced in full.

C8 Cross-domain validation and falsifiable evaluation (Sections 11, 12 and 15). Five projects traced end-to-end, a complete problem-to-mechanism map, and eight hypotheses with explicit rejection criteria.

2 The problems of multi-actor work

The problems below recur in research, events, construction, software and every other field where several actors work toward one result. One person with agents meets most of them too, because the agents are actors. Section 3 gives the evidence for each, and Section 12 maps each to the mechanism that addresses it.

P1 Aims that nobody can check. A group agrees on an aim in words, such as a strong paper, a smooth wedding or a house that passes inspection, and each member reads those words differently. Without an agreed test of completion and a named judge, work drifts until disagreement surfaces at delivery.

P2 Promises that nobody can see. Who promised what to whom, and by when, sits buried in conversations, email and contracts. When someone leaves, falls ill or forgets, the obligation vanishes from view until downstream work stalls.

P3 Waiting that nobody can see. Work stops while an actor waits on a decision, a delivery, a review or a permit. Queues of blocked work rarely appear on schedules, even though they dominate end-to-end cycle time in product development [94]; an agent that waits typically polls or sleeps invisibly.

P4 Status that has to be asked for. The state of the work is kept current nowhere, so people ask for it, report it and meet to align it. Surveys of knowledge workers put the share of the working day spent on such work about work at more than half (Section 3.1).

P5 Dependencies found late. A task needs the result of another: an analysis needs cleaned data, the caterer needs the final guest count, the ceiling needs the ducts installed first. A dependency that nobody stated is found when the work that needs it stops.

P6 Changes that collide. Two actors change the same thing, or one changes what another's work depends on. Two authors rewrite one section of a grant proposal, two trades route a pipe and a duct through the same ceiling space, two people book one venue for the same evening, two agents edit one file. The collision is usually found late, sometimes after the work is built.

P7 Decisions without a record or a right. Decisions are made in meetings and chats, with no record of what was considered or of who had the right to make them. They get made twice, reversed by someone without the authority, or forgotten.

P8 Evidence apart from the claims it supports. Test results, inspection reports, stamped drawings, receipts and photographs sit in folders apart from the claims they support, so nobody can tell whether a claim still holds after the thing it is about has changed.

P9 Work closed by the actor who did it. When the actor who did the work decides it is done, it is marked done before the actor it was for has accepted it. A measure that becomes a target stops being a good measure [105], and agents in particular report completion they did not achieve (Section 3.7).

P10 Trust, credentials and liability. Some work may be done or accepted only by a qualified party. A licensed engineer stamps a structural drawing, a certified electrician signs off wiring, an ethics board approves a study, an officiant registers a marriage. The check that the right party acted is manual, and an agent acting for someone has no explicit limits on what it may do.

P11 Shared capacity. People's hours, machines, rooms, cranes, compute and budgets are shared. Double bookings and idle capacity both go unseen until they cost time.

P12 Work across organisations. Clients, contractors, vendors and partners keep their own tools. What they promised each other lives in contracts and email, changes arrive as documents, and each side sees only part of the state.

P13 Plans that are not kept. People underestimate how long their own work will take [18], and most large projects exceed their budgets or schedules (Section 3.2). A plan that is not revised against what happened soon stops describing the work.

P14 Work given to the wrong actor. People spend time on tasks that an agent does well, agents are given tasks that need judgment or accountability, and experts spend paid hours on clerical work. The boundary between what AI systems do well and what they do badly is jagged, and it moves (Section 3.6).

P15 Agents that coordinate badly. Groups of agents fail in recurring ways: unclear specifications, misalignment between agents, and weak or missing verification (Section 3.7). Agents can also be steered by text they read [54], and they spend time and money without limit when nobody set one.

None of these problems belongs to one discipline. Each has been studied in some field: coordination theory, construction management, the science of team science, the economics of platforms, and the study of people working with AI. Section 3 brings that evidence together.

3 Evidence from research, experiments and industry

This section gathers what is known about the problems of Section 2. It draws on coordination theory, project management, construction, the study of research teams, experiments with people and AI systems, studies of agents working together, and the economics of platforms for work. Industry surveys appear where they add a measurement that research lacks, and their sources are named, since several come from vendors of collaboration tools.

3.1 Coordination is much of the work

Malone and Crowston defined coordination as managing dependencies between activities, and they catalogued the mechanisms that manage each kind of dependency: shared resources, prerequisites, simultaneity and the relation between tasks and subtasks [75]. Brooks observed that adding people to a late software project makes it later, in part because the paths of communication grow with the square of the team's size [15]. Herbsleb and Mockus found that work items spread across sites took about two and a half times as long as similar items done at one site, mostly because of delays in communication [58].

Surveys of knowledge workers measure the same cost directly. In Asana's survey of about 9,600 workers, respondents spent 58% of their working day on work about work, such as communicating about work, searching for information and switching between applications, 33% on the skilled work they were hired for, and 9% on strategy [6]. Microsoft's measurement of activity in its own applications found that the average user spent 57% of their time communicating in meetings, email and chat, and 43% creating documents, spreadsheets and presentations [79]. Both come from vendors of collaboration tools and measure self-reports or tool activity. Their agreement with each other and with the older research is what matters here: in groups, keeping one another informed costs as much time as the work itself.

3.2 Large projects miss their plans

Flyvbjerg and Gardner report that in a database of more than 16,000 projects of many types, 8.5% met both their budget and their schedule, and 0.5% met their budget, their schedule and their promised benefits [45]. A study of more than 5,400 information-technology projects with budgets above $15 million found that they ran on average 45% over budget and 7% over time, and delivered 56% less value than predicted [13]. The Project Management Institute's survey of 2020 found that organisations wasted on average 11.4% of their investment through poor project performance [87]. Individuals show the same bias in their own work: people predict that their tasks will finish sooner than they do, even when they know that similar tasks ran late [18]. The causes of overruns are many, and optimism in estimates and the size and novelty of projects matter as much as coordination [44]. What an engine for coordination can do is show the gap between plan and reality while there is still time to act on it.

3.3 Reliable promises make work flow

Winograd and Flores described organisations as networks of commitments made in language: a request, a promise, a report of completion and an acceptance [118]. Construction put the idea into practice. In the Last Planner System, the people who will do the work promise each week what they will complete, and they promise only tasks whose prerequisites are ready. Reliability is measured as Percent Plan Complete, the share of those promises kept, and the reason for every broken promise is recorded and acted on [9]. Macomber and Howell grounded the system in the theory of linguistic action, as a practice for making and keeping promises reliably [73]. It is a core practice of lean construction.

The effect holds for individuals as well. A meta-analysis of 94 tests found that if-then plans, which say when and where a person will act, had a medium-to-large effect on reaching goals, with an effect size of $d = 0.65$ [52]. In an experiment on deadlines, people who set their own deadlines did better than those with none, and people given evenly spaced deadlines did better still [5]. Explicit promises with dates, made by those who will do the work and checked at a steady cadence, make work more predictable for a team and for one person alike.

3.4 Shared models catch conflicts before they are built

Building information modelling keeps a building's walls, beams, ducts, pipes and cables in one typed model that every discipline works on. Because the model is to scale in three dimensions, the systems of different trades can be checked automatically for clashes before anything is built [8]. Stanford's Center for Integrated Facility Engineering reported,

across 32 projects, up to 40% elimination of unbudgeted change and up to 7% reduction in project time [8]. ISO 19650 standardises how such information is managed across the life of an asset [63], and the Industry Foundation Classes give the model an open schema [62].

Software shows the same pattern at a finer grain. Cataldo and colleagues computed which developers needed to coordinate from the files their tasks touched, and found that tasks finished faster when the developers' communication matched those requirements [21]. Code-owner files name who must approve a change to each part of a repository [50]. When the things the work is about are kept in a shared model, conflicts between actors can be computed from what each actor's work touches, and they can be found before the work is done.

3.5 Teams carry knowledge work, and coordination limits them

Wuchty, Jones and Uzzi analysed 19.9 million papers and 2.1 million patents over five decades and found that teams increasingly produce knowledge in every field, and that work by teams is more often highly cited than work by individuals [122]. Collaboration across institutions has a cost. In 491 research collaborations funded by the US National Science Foundation, projects that spanned more universities carried out fewer coordination activities and reported fewer outcomes, and coordination, especially dividing responsibility for tasks and transferring knowledge among investigators, predicted results [28]. Neuromatch Academy, an online summer school, ran its first edition for 1,757 students and 191 teaching assistants in small groups across time zones, with research projects carried out in those groups [115]. Distributed research teams of this kind form quickly, work for weeks, and depend on the clear division of responsibility that the study found predictive.

3.6 People with AI gain inside its competence and lose outside it

Field experiments show substantial gains when AI systems handle tasks inside their competence. Among 5,172 customer-support agents, an AI assistant raised issues resolved per hour by roughly 15%, helping novices most [16]. On professional writing tasks, ChatGPT cut completion time by 40% while raising output quality by 18% [82], and developers using an AI pair programmer finished one well-defined coding task 55.8% faster [86]. Among 758 consultants, AI assistance raised task completion by 12.2%, speed by 25.1%, and quality inside the model's competence. On a task designed to fall outside it, assisted consultants were 19 percentage points less likely to reach a correct answer [29].

Gains depend sharply on task structure and baseline expertise. In a randomised trial with 16 experienced open-source maintainers across 246 tasks in repositories they knew well, early-2025 AI tools made them 19% slower, even though they had forecast a 24% speed-up and believed afterwards that they had been 20% faster [10]. Across 106 experiments, human–AI combinations underperformed the stronger solo side on average, losing ground on decision tasks while gaining on creative tasks and whenever human expertise exceeded the model's [111]. Across the wider economy, roughly 80% of US workers have at least a tenth of their tasks exposed to language models, and 19% at least half [36]. An engine must therefore decompose work at the task level, route each task by comparative advantage, verify outputs independently, and keep final judgment with an accountable actor.

3.7 Agents working together fail in recurring ways

Cemri and colleagues annotated more than 1,600 execution traces from seven frameworks for systems of several agents and found 14 failure modes in three categories: system design issues, misalignment between agents, and task verification [22]. In a simulated software company where agents browse, write code and talk to simulated colleagues, the best agent completed 30.3% of tasks on its own [123]. In a benchmark of agents that serve users under a domain's rules, state-of-the-art agents succeeded on fewer than half of the tasks, and in the retail domain fewer than a quarter of tasks succeeded in all eight of eight repeated trials [124]. Text that an agent reads can carry instructions that steer it [54]. Current frameworks coordinate agents through conversations, assigned roles or graphs of steps [59, 71, 88, 121], and they leave promises between agents, the evidence for them and their acceptance to the application. Agents need what people need, more strictly: a stated specification, obligations others can see, verification by someone other than the agent that did the work, and grants and budgets that bound what they may do.

3.8 Markets for work are growing and need structure

The International Labour Organization counted 142 online and location-based platforms for work in 2010 and more than 777 in 2020 [61]. Economics explains why. Coase argued that firms exist because using a market has costs [25], and Malone, Yates and Benjamin predicted that information technology, by lowering the cost of coordination, would shift activity from hierarchies toward markets [76]. Markets today handle small, well-defined tasks best. Kittur and colleagues set out what crowd work needs before it can carry complex work: decomposition of tasks, coordination among workers, quality assurance, and fair pay and careers [68]. Flash teams and flash organisations then assembled paid experts from online markets into structured teams that completed complex, open-ended projects [96, 112]. Studies of platform workers show the risk on the other side: autonomy under algorithmic control, with overwork and isolation for some [119], and information that the platform holds and the worker lacks [97]. A market for complex work needs terms that both sides can see, acceptance by the client against evidence, a reputation that both sides can check, and protection against control by one side.

3.9 What the evidence asks of an engine

Eight requirements follow. They are the measure against which the design of Sections 5 to 9 should be judged.

R1 Every promise is explicit, with a debtor, a creditor, a date and a test of completion, and every broken promise carries its reason (Sections 3.2 and 3.3).
R2 Waiting and status are visible to everyone without anyone asking (Section 3.1).
R3 Dependencies and conflicts are computed from what each piece of work touches, as early as possible (Sections 3.4 and 3.5).
R4 Work is accepted by the actor it is for, against evidence that stays attached to the claim it supports (Sections 3.6 and 3.7).
R5 Work goes to the actor best placed to do it, and judgment stays with the actor accountable for it (Section 3.6).
R6 Agents act under grants and budgets, and their work is verified by another actor (Section 3.7).
R7 Credentials and decision rights are explicit and can be checked (Sections 3.4 and 3.8).
R8 Coordination works across organisations and in markets, with records that every party can check (Sections 3.5 and 3.8).

4 Division of labour: people, agents and experts

4.1 Tasks, comparative advantage and hand-offs

Automation has historically reshaped occupations more often than it has erased them, replacing routine tasks while raising the return to complementary human judgment [7]; employers surveyed by the World Economic Forum project 170 million roles created alongside 92 million displaced between 2025 and 2030 [120]. Because the task is the unit that shifts across the jagged frontier of Sections 3.6 and 3.7, projects divide across four kinds of participant by comparative advantage:

- *Agents* are fast, inexpensive at scale and continuously available. They excel at broad search, drafting, schema transformation and checking stated rules, yet remain brittle over long horizons, variable across runs, and legally unaccountable unless a principal answers for them.
- *People* judge under ambiguity, set priorities, negotiate trade-offs and bear moral and contractual responsibility, though their attention is scarce and every interruption carries a cognitive switching cost [60].
- *Experts and certified actors* contribute scarce specialist skill and statutory authority, such as a structural engineer's stamp, a building inspector's sign-off, an ethics board's approval or an auditor's opinion, backed by professional liability and insurance.
- *Organisations* hold capital, long-lived contracts and institutional liability, delegating operational authority to members and agents through bounded grants.

Dividing tasks across actors pays off only if the hand-offs cost less than the specialisation saves. Whenever an agent drafts a permit package for an architect, a student hands a cleaned dataset to a co-author, or a subcontractor hands a roughed-in zone to the next trade, unreviewed drafts and unrecorded decisions become the primary source of rework. Human–AI interaction guidelines ask single-user tools to make capabilities explicit, explain actions and make correction cheap [2]; a multi-actor engine must lift those same guarantees to explicit hand-offs between arbitrary actors.

4.2 Project-scoped organisations and verifiable trust

Lower coordination costs shift production from permanent hierarchies toward fluid, project-scoped teams and markets [25, 74, 76]. Flash organisations showed that online specialists assembled into dynamically reconfigurable roles can deliver complex, open-ended engineering and design work [112], while recent enterprise surveys report early human–agent "frontier" teams in which individuals direct fleets of bounded agents [80]. Because such teams assemble rapidly, rotate specialists and dissolve on delivery, project state must outlive any participant, so that a newcomer, whether a person or an agent, can reconstruct the context by query without interviewing the others.

Coordinating strangers, external contractors and autonomous agents requires trust that a machine can check, since the parties do not know one another. Verifiable credentials allow licensing boards, universities and employers to issue signed, time-bounded claims of qualification [104]; provenance graphs record who derived which artifact from which inputs [81]; and explicit grants cap what an agent may touch or spend on its principal's behalf. Combined with escrowed payment on creditor acceptance and portable, trust-weighted delivery histories, these mechanisms are meant to satisfy the eight requirements (**R1**–**R8**) of Section 3.9 without centralising asymmetric control in a platform operator.

5 The work graph

Consider one promise on a building project (Figure 1). The electrical subcontractor (Actor) promises the general contractor (Actor) that cable tray T-4 in the level-3 ceiling void will be installed and free of spatial clashes (Proposition) by 21 March, reserving two days of scissor-lift access (Resource) under its subcontract grant (Grants). When the subcontractor uploads photographs and runs the 3D clash check, those outputs enter as immutable Records linked to the proposition by Attests; when the inspector signs off, a Decides assertion discharges the Commits edge. Tray T-4, the

nearby duct D-12 and beam B-7 stay outside the work graph: they live in the building model. To state how promises interact with the artifacts they govern, we begin with that underlying store.

5.1 Subject spaces and the joint predicate space

Work is always about something: a manuscript, a dataset, a venue calendar, a building model, a software system. The work graph references these artifacts while leaving them in the *subject space* that already stores them. Because commitments can also reference other commitments, grants and resource capacities, admission reasons over the disjoint union $\Pi = \Pi_{\text{sub}} \sqcup \Pi_{\text{wg}}$ of subject-space predicates and work-graph predicates over joint states $s = (s_{\text{sub}}, s_{\text{wg}})$.

Definition 1 (Predicates, write sets and read sets)**.** A *versioned state space* has states, identified by revisions, made of *entities*: the elements of the subject space (files, directories, beams, bookings, typed model elements) and of the work graph (actors, propositions, records, resources and assertions). For every entity x the space distinguishes three kinds of *predicate*. The *existence* of x says whether x is present. The *extent* of x is its content and attributes and its place in the structure: the directory of a file, the owner of a port, the parent of a node, the scope of an assertion and, for an assertion, the entities it ends on.The *dependents* of x are a family of *membership* predicates, one for each entity y that could be attached to x, each saying whether y is attached: a file in a directory, a port that x owns, a child of x, an assertion that ends on x. These are the *structural* predicates, and a state is determined by their values. In addition, for a resource r, an amount k and an interval I, the *threshold predicate* $\pi_{r,k,I}$ holds at s when at least k of r is unreserved throughout I, written $\text{avail}(r, I, s) \geq k$. Threshold predicates are read only when a reservation is tested; a commitment that needs an amount to stay free holds a reservation for it. We write $\Pi = \Pi_{\text{sub}} \sqcup \Pi_{\text{wg}}$ for the set of all predicates, split by the space their entity belongs to, and say that two states *agree on* $S \subseteq \Pi$ when every predicate of S has the same value in both.

A *change* P maps a state s to a state $P(s)$. Its *write set* $Wr(P, s)$ consists of the predicates that P sets: the existence and the extent of every entity it creates, removes or replaces, the membership of every entity it attaches or detaches, and every predicate whose value is computed from the state (a threshold predicate, or the index predicates of Section 7) that P changes. A set $R \subseteq \Pi$ is a *sufficient read set* for a query q at s if q gives the same answer at every state that agrees with s on R, and it is *stable* if evaluating q at any such state reads exactly R again. We write $Rd(q, s)$ for the read set of q at s. Checks, acceptance queries and antecedents read structural predicates only.

The predicates are chosen so that read sets are as small as they can be without missing a conflict. Following an assertion to the entity it ends on reads only the existence of that entity, so it does not conflict with a change to the entity's extent. Listing what is attached to x reads the whole family of its dependents, so it conflicts with any later attachment: a check that found no reader of a dataset column is invalidated by the arrival of the first reader, which is the case databases call a phantom [38]. Attaching an entity to x sets one member of the family and reads none, so two changes that each attach something to x do not conflict with each other. Threshold predicates do the same for quantities: two reservations of one dollar each from a pool of a thousand both read the predicate "at least one dollar is free", which neither of them makes false, so they do not conflict [84].

Subject spaces differ in the granularity of Π_{sub} and in how $Rd(q, s)$ is obtained. Over a shared drive or Git repository, the entities are files and directories, the dependents of a directory are its listing, and read sets are *declared* through test selectors or code-owner rules [50]. A declaration that does not depend on the state is stable, and it is sufficient only if it is complete; declarations can omit hidden dependencies unless hermetic sandboxing enforces them [108]. In a booking calendar, the entities are room–slot pairs; in a lineage-tracked data warehouse, tables and column versions. In an open-schema building model (IFC) [62], the entities are single beams, ducts and zones, and geometric clash checks record the elements they inspect. In a typed system model such as Reaktor's system graph (Section 13.1), the store itself records every predicate a traversal inspects. A *recorded* read set is sufficient by construction, and it is stable because a deterministic evaluation that has read the same values so far makes the same next read (Sections 10 and 13).

5.2 Kinds and reified assertion families

Group work requires five categories of shared coordination state: outcomes, actors with their capacities, commitments, evidence, and authoritative decisions. The work graph represents all five using four element kinds and six assertion families (Figure 1). Every assertion is a *reified, addressable hyperedge* with a store-issued identity, allowing assertions to reference other assertions directly without wrapper nodes:

- An Actor is a person, team, organisation, AI agent or service account; every actor acts solely under grants.
- A Proposition is a statement whose truth value (*holds*, *fails* or *unknown*) is computed by a query: a guest count is final, a beam carries its design load, an analysis reproduces from raw data, p95 latency stays below 100 ms. Outcomes, consequents, antecedents, hypotheses, contract terms and decision choices are propositions.
- A Record is immutable, content-addressed by digest, and stamped with source and time: a progress report, delivery receipt, test log, lab certificate, signed credential or message.
- A Resource is a bounded capacity consumed or reserved over time: specialist hours, a tower crane, a venue date, a test device pool, a compute budget or escrowed funds.

The six assertion families connect elements and other assertions by named roles:

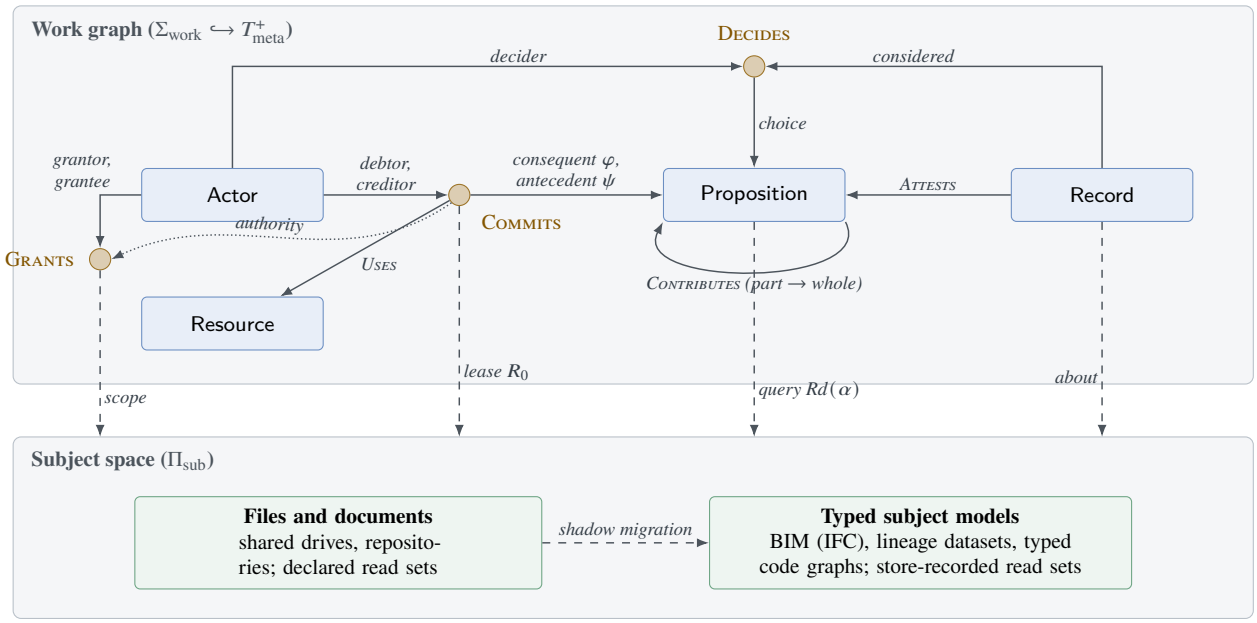

Figure 1: The work graph and its subject space. Four element kinds (blue boxes) join through six addressable assertion families; multi-role assertions (COMMITS, DECIDES, GRANTS) appear as amber hyperedge nodes, and reified incidences let COMMITS and DECIDES cite GRANTS while USES cites COMMITS. Dashed arrows cross into the subject space: grant scopes, declared write leases R_0, proposition queries $Rd(\alpha)$ and record subjects. A project can migrate online from file-level declarations to a typed model (Section 10) or co-reside in one unified hypergraph store (Section 13).

- COMMITS $(d, r, \varphi, R_0, \psi, g, \kappa, t, h, \alpha, \tau)$ binds debtor d and creditor r to consequent φ under authority grant g (Section 6).
- ATTESTS (evidence, proposition, verifier) links a Record to a Proposition together with the verdict of an actor or automated check.
- DECIDES (decider, choice, authority, considered, supersedes) records that an actor adopted a proposition under grant `authority`, citing the records considered and optionally superseding an earlier decision. Each DECIDES assertion is immutable, though a subsequent authorised decision may supersede it.
- GRANTS (grantor, grantee, right, scope, condition, expiry) confers a bounded right, subject to a credential proposition `condition` and an optional `expiry`. The right may be to write into a scope, to decide, to accept the discharge of a commitment, to reserve a resource or to act on behalf of the grantor.
- CONTRIBUTES (part, whole) states that achieving or maintaining `part` contributes toward `whole`. An *outcome* is a root proposition adopted via DECIDES.
- USES (commitment, resource, amount, interval) reserves `amount` of a Resource over `interval` for a COMMITS assertion.

Malone and Crowston's taxonomy of coordination dependencies [75] maps onto these primitives: shared resources become multiple USES on one Resource; producer–consumer prerequisites become antecedents ψ; simultaneous collaborative action becomes a team Actor or merge-law region; and task decomposition becomes delegated commitments linked by CONTRIBUTES. In Section 13, we show that these four kinds and six families form the coordination sub-signature $\Sigma_{\text{work}} \hookrightarrow T^{+}_{\text{meta}}$ of one unified meta-model.

5.3 Actors, grants, revocation and trust-weighted capabilities

Every actor, whether a person, an agent, an expert or a firm, obeys the same rules: act under grants, form commitments, attach evidence and discharge obligations through creditor acceptance. Three constructions capture their institutional differences.

Delegated agency and revocation. A team is an Actor whose members hold GRANTS to act for it. An AI agent acts for a person or firm under a grant bounded by region scope, resource budget and `expiry`; commitments made within that grant bind the principal just as an agent binds a principal in agency law. A grantor may supersede or revoke a grant at any revision. If an agent's grant is revoked, or an expert's licence expires, while a commitment c relying on it is *active*, admission clause (A3) refuses further writes under that grant, c is marked *at risk*, and the grantor and the creditor are notified. Obligations already entered on a principal's behalf remain binding until they are amended or released.

Verifiable credentials. A credential is a signed Record issued by an authority (a state licensing board, university or ethics committee) attesting to a qualification proposition about an actor until a stated expiry [104]. Because GRANTS and

acceptance queries can require that proposition to hold at admission time, statutory sign-offs such as stamping structural calculations, certifying electrical work or registering a marriage are enforced by the store at admission, and nobody has to remember a checklist.

Sybil-resistant capabilities. An actor's track record is a query over discharged and violated commitments, their attached evidence and creditor acceptances. In an open network (Section 9.3), unweighted counts would invite Sybil wash-trading, in which an actor mints fake client identities and accepts trivial commitments from itself [23]. Capability queries therefore evaluate histories *relative to the viewer's trust roots*, weighting commitments by creditor credential verification, transitive trust from known institutions, or settled external resource transfers.

5.4 The four questions as graph queries

The four questions of Section 1 compile to read-only traversals:

- *What is the group trying to achieve?* Active outcomes adopted by un-superseded Decides assertions, annotated with rollup status across their Contributes subgraphs.
- *Who owes what to whom?* Every *conditional* and *active* commitment with its debtor, creditor, consequent, modality, deadline, review state and *at risk* flag.
- *What evidence changed the plan?* Every amendment, superseding decision, release and cancellation ordered by revision, joined to the Records cited via Decides and Attests.
- *Who can decide next?* Every pending acceptance review, open decision, refusal escalation or violated commitment joined to actors holding matching Grants, ordered by the cost-of-delay rank of Section 8.

Because status is a query over the current state of the graph, nobody maintains status slides, and people, agents and external partners read one consistent answer.

6 Commitments and verbs

Definition 2 (Commitment)**.** A *commitment* is an addressable Commits assertion

$$c \;=\; (d,\, r,\, \varphi,\, R_0,\, \psi,\, g,\, \kappa,\, t,\, h,\, \alpha,\, \tau)$$

comprising:

- a *debtor* d and a *creditor* r, both Actors (individuals, teams, agents or organisations);
- a *consequent* φ and an *antecedent* ψ, both Propositions;
- a *declared lease region* $R_0(c) \subseteq \Pi$ naming the predicates the debtor's work directly mutates, together with an *authority* grant g empowering d to write into $R_0(c)$;
- a *temporal modality* $\kappa \in \{\mathit{achieve}, \mathit{maintain}\}$ distinguishing one-off delivery ($\Diamond\varphi$) from a standing invariant ($\Box\varphi$);
- a *deadline* t (with $t = \infty$ permitted for open-ended *maintain* invariants), a *heartbeat interval* h, an optional creditor *review window* Δ_{rev}, and an *acceptance query* α agreed by r that decides whether φ holds;
- a *timeout policy* $\tau \in \{\mathit{fail\text{-}closed}, \mathit{optimistic\text{-}at\text{-}risk}\}$ governing admission when α exceeds the synchronous check budget (Section 7).

Its lifecycle state lies in $\{\mathit{proposed}, \mathit{conditional}, \mathit{active}, \mathit{discharged}, \mathit{violated}, \mathit{released}, \mathit{cancelled}, \mathit{declined}\}$, while an *active* commitment additionally carries two orthogonal status flags: *at risk* (set when a heartbeat lapses) and *in review* (set when a timely delivery report awaits the creditor's decision within Δ_{rev}). At state s, the *observed guard region* and *total region* of c are

$$R_{\text{obs}}(c, s) \;=\; Rd(\alpha, s) \;\cup\; Rd(\psi, s), \qquad R(c, s) \;=\; R_0(c) \;\cup\; R_{\text{obs}}(c, s).$$

Both regions contain structural predicates only. A *guard* is a standing commitment in which the debtor and the creditor are the same actor and the declared region is empty: $d = r$, $R_0(c) = \emptyset$, $\kappa = \mathit{maintain}$. It states that something must stay true and names an owner to ask.

The two regions have different jobs. In the ceiling void of Figure 3, the electrical subcontractor's commitment c_E declares $R_0(c_E) = \{\text{tray T-4}\}$, and its clash check α_E inspects every element in the void, placing duct D-12 and beam B-7 inside $R_{\text{obs}}(c_E, s)$. Likewise, a standing commitment bounding frame time in a game or p95 latency in a service reads much of the codebase through R_{obs} while owning no exclusive write lease ($R_0 = \emptyset$). The exclusive lease of the debtor (Definition 3, A4) covers only $R_0(c)$, and only while c is *active*. Over $R_{\text{obs}}(c, s)$, clause A5 lets any authorised actor change surrounding elements as long as α does not go from holding to failing.

Bilateral consent and temporal modalities. Figure 2 shows the state machine, extending multi-agent commitment protocols [101, 125] with bilateral consent, standing invariants and creditor review protection. No actor can unilaterally bind another: moving from *proposed* to *conditional* (when ψ does not yet hold) or directly to *active* (when ψ already holds) requires both d's `commit` under grant g and r's consent to the terms (implicit when r authored the proposal, explicit via `decide` when d proposed an offer). Once *active*, discharge and violation follow κ:

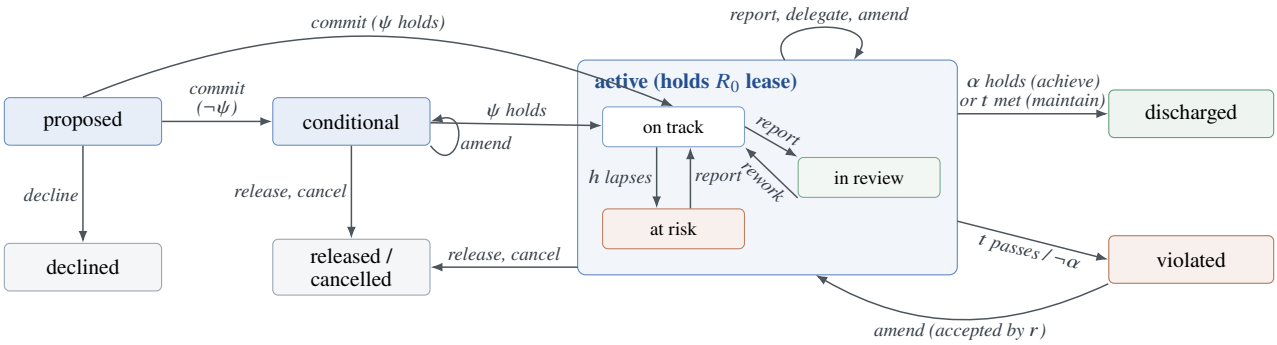

Figure 2: Lifecycle of a commitment. Bilateral `commit` moves a proposal to *conditional* or directly to *active*. While *active*, the debtor holds the R_0 lease; a lapsed heartbeat sets *at risk*, and a timely delivery `report` awaiting the creditor's `decide` enters *in review*, which pauses the debtor's violation clock for Δ_{rev}. Achievement commitments discharge when α holds and violate if t passes first; standing (*maintain*) commitments stay active while α holds and violate immediately on regression.

Verb	State transition and precondition	Example
propose	creates c in state *proposed* recording its proposer; any actor within scope may propose	a caterer offers to serve 150 guests on 14 June at a stated price
commit	moves *proposed* c to *conditional* ($\neg\psi$) or *active* (ψ), acquiring $R_0(c)$ if active; requires d's authority g and r's consent	a licensed structural engineer commits to check and stamp bay drawings by Friday
decline	moves *proposed* c to *declined* with a recorded reason	a photographer declines the date, freeing the planner at once
delegate	creates a child commitment c' ($d' \to d$) whose consequent contributes to φ; d remains answerable to r	a student delegates literature screening to an agent under a token cap
report	links a Record via Attests to φ, clears *at risk*, renews h, and enters *in review* if awaiting r's decision	a contractor logs concrete cylinder test results and delivery tickets
decide	records an immutable Decides assertion under a decision grant, citing considered records	the couple selects a venue, citing three competing quotes
amend	updates φ, R_0, t, κ or reassigns d (novation) on *conditional*, *active* or *violated* c; requires consent of r and bound d	after a design change, client and contractor extend a milestone by one week; an on-call shift hands over a standing invariant
release	creditor r frees debtor d; c moves to *released* and drops any active lease	a mentor drops an exploratory analysis the paper no longer needs
cancel	debtor d withdraws with recorded notice; c moves to *cancelled* (subject to contract terms)	a vendor withdraws within its contractual notice window
escalate	routes a *violated*, *at risk* or stalled commitment to r or a named arbiter with a witness	an agent blocked on an ambiguous zoning clause escalates to the architect

Table 1: The ten commitment verbs. Every verb compiles to an admitted graph patch executed identically for people, agents and organisations.

- For κ = *achieve*, c becomes *discharged* as soon as α holds prior to t, and *violated* if t passes without α holding. There is one exception: when d submits a delivery `report` before t and α awaits r's `decide`, c enters *in review*, which pauses d's violation clock for Δ_{rev}, so a slow creditor never makes a punctual debtor late.
- For κ = *maintain*, c stays *active* while α holds, is discharged when a finite horizon t is reached unbroken (or on `release` when $t = \infty$), and transitions immediately to *violated* if any state change or imported observation falsifies α.

6.1 Verbs

Ten domain verbs evolve commitments (Table 1), covering Winograd and Flores's conversation for action [118] alongside Singh's social commitment operations [101]. The verbs are distinct from the store's transport of patches (`submit`, Section 13): each verb constructs a patch, and every patch passes the same admission gate (Section 7).

7 Admission against commitments

Every change to the subject space or the work graph passes through one admission gate. It is the same gate for a person editing a specification, an agent reserving budget and a subcontractor rerouting a duct.

A change reaches the store as a *patch*: a finite list of structural assignments (create or remove an entity, replace an extent, attach or detach an entity) and of reservations, together with a base revision v, an author and a grant. A patch names what it removes and what it replaces, so it reads the existence and the extent of every entity it changes. It never leaves a dangling attachment: it removes an entity only together with everything attached to it, which it lists, so it reads

the dependents of what it removes. Entities that it creates take identifiers that belong to the patch alone. Over a typed graph, patches are the double-pushout rules of Section 13.1.

To find the commitments a patch concerns, the store keeps two *index* predicates for each structural predicate π and each commitment c, both computed from the state: the *lease index* $\lambda_{\pi,c}$ holds when c is *active* and $\pi \in R_0(c)$, and the *guard index* $\gamma_{\pi,c}$ holds when c is *conditional* or *active* and $\pi \in R(c, s)$. For a fixed π they form two families, one member per commitment. A patch that creates, amends or ends a commitment, or that changes what its queries read, sets the members whose values change.

The *read set* $Rd(P)$ of a patch P consists of: (i) the predicates read to construct P and to decide whether it applies; (ii) the read sets of every check evaluated on its result; (iii) the read sets of the acceptance query and the antecedent of every commitment that admission evaluates, before and after the change, and of every commitment that P creates or amends, on its result; (iv) the predicates that select the grants, credentials and budgets admission consults, the threshold predicates of the reservations in P, and, for every structural predicate that P sets, the whole lease and guard index families of that predicate together with the extents of the commitments they name. Item (iv) is what stops a change in flight from slipping past a promise formed after its base: forming the promise sets a member of an index family that the change read.

Definition 3 (Admission)**.** Let P be a patch by actor a under grant g, with base revision v, presented when the head revision is v'. The store admits P if and only if five clauses hold:

(A1) *Validity:* P applies at v', the result $P(v')$ is well formed (for a graph, the axioms of Definition 4), and $P(v')$ passes the structural checks of the subject space, such as syntax and test gates, geometric well-formedness or slot availability;

(A2) *Freshness:* no patch admitted between v and v' set a predicate of $Rd(P)$;

(A3) *Authority and escrow:* grant g is unrevoked and unexpired at v', its credential conditions hold, g covers the structural predicates that P sets, and every Uses reservation in P fits both within the resource's remaining capacity at v' and within g's remaining budget, each read through threshold predicates;

(A4) *Write lease on R_0:* for every *active* commitment c at v' such that P sets a predicate of its declared region $R_0(c)$, either a is c's debtor or an active delegate, or c's terms admit P under a stated merge law, or P carries a Decides permission from c's creditor, or P is a consented amendment of c;

(A5) *Semantic guard on $R(c, v')$:* for every *conditional* or *active* commitment c at v' such that P sets a predicate of $R(c, v') = R_0(c) \cup R_{\text{obs}}(c, v')$, P does not turn α from *holds* at v' to *fails* at $P(v')$, does not falsify a *maintain* invariant, and does not make an antecedent ψ that held fail, unless P amends or releases c. A patch that records an observation of the world is admitted even when it falsifies α, and c then follows its lifecycle (Section 6). If evaluating α or ψ exceeds the synchronous budget, c's timeout policy τ decides: under $\tau =$ *fail-closed*, the default for regulated, structural and financial commitments, the patch is refused for now and placed in an asynchronous verification queue; under $\tau =$ *optimistic-at-risk*, P is admitted, c is flagged *at risk*, and d and r are notified.

Every refusal returns the failed clause together with a *witness* (c, π, d, r) naming the blocking predicate, the commitment and its holder. A mechanical subcontractor whose rerouted duct clashes with tray T-4 passes (A4), since the duct lies outside $R_0(c_E) = \{\text{T-4}\}$, and fails (A5) with a witness naming c_E and the electrical subcontractor. Moving the same duct 5 mm without a clash writes into $R_{\text{obs}}(c_E, v') \setminus R_0(c_E)$ while α_E stays true, so both (A4) and (A5) admit it at once. A conditional commitment holds no lease on R_0 until its antecedent holds, so distant future phases do not freeze today's workspace.

A *merge law* permits concurrent writes inside a shared R_0 without a round trip of coordination: append-only evidence logs, RSVP sets and additive counters commute as state- or delta-based CRDTs [99], and live co-editing sessions merge by operational transformation or sequence CRDTs [35]. A merge write replaces an extent without reading it, and the merge law is what makes such writes commute; the theorem below does not cover them. A merge law converges, and convergence alone does not keep an invariant over the merged value true, so clause (A5) still checks every guard whose region a merge writes.

Throughout, queries are evaluated deterministically, and whether an evaluation stays within the synchronous budget depends only on the values it reads.

Theorem 1 (Independence)**.** *Let P_1 and P_2 be patches with base v that are each admissible at v, whose read sets are sufficient and stable, and suppose that neither sets a predicate that the other reads:*

$$Wr(P_1, v) \cap Rd(P_2) = \emptyset \qquad \text{and} \qquad Wr(P_2, v) \cap Rd(P_1) = \emptyset.$$

Then P_2 is admissible after P_1, P_1 is admissible after P_2, and both orders produce the same state.

The hypothesis does not mention two patches that set the same predicate, because it already excludes the harmful cases: a patch reads whatever it removes or replaces, so two patches that satisfy the hypothesis set no structural predicate in common (Appendix A). Two patches that each attach an edge to one hub satisfy it, since each sets one member of the hub's dependents and reads none. Two reservations of k_1 and k_2 from a resource with $\text{avail}(r, I, v) \geq k_1 + k_2$ satisfy it too, since neither makes false the threshold predicate that the other reads. Two agents editing disjoint modules while each

reserving $1 from a $1,000 compute pool are therefore independent while at least $2 remain, and both commit without a retry. When their combined demand exceeds what remains, the first to commit makes false the threshold predicate that the second read, and the second is refused under (A2) or (A3). For the graph structure alone, the equality of the two results is the Local Church–Rosser theorem for parallel independent rewrites [34]. The theorem adds that well-formedness, authority, leases and guards survive as well, which the rewriting theorem by itself does not give.

The rule that only reservations read threshold predicates is needed for the theorem. Suppose 10 units are free, a guard requires that at least 3 stay free, and two patches reserve 3 and 7. Each is admissible alone, and neither changes a threshold predicate that the other reads, yet together they leave nothing free. A commitment that needs 3 units kept free therefore holds a reservation of 3, which leaves 7 free. The patch that reserves 7 then reads "at least 7 are free", which the patch that reserves 3 makes false, so the two are not independent and the second to commit is refused.

Proposition 2 (The history is serialisable)**.** *Let P have base v, let the head be v', and suppose clause (A2) holds. Then every query evaluated while P was constructed has the same answer at v' as at v, every check and commitment query evaluated for P has the same answer on $P(v')$ as on $P(v)$, and admission consults the same commitments at v' as at v. Two consequences follow.*

(a) The gate may evaluate the checks and guards of a patch on a snapshot at its base, with no lock held, and admit the patch after repeating only the freshness test at the head.

(b) The sequence of admitted patches has the same result as carrying out the changes one at a time in the order of admission, each constructed on the result of the last.

Consequence (b) is serialisability. It is stronger than what databases give under snapshot isolation, which lets two transactions that each preserve an invariant break it together [11, 42]. Here the check of the invariant is in the read set of both patches, so the second is refused.

Corollary 3 (Independent flows)**.** *Let two tasks be such that no patch of either sets a predicate that any patch of the other reads. Then an interleaving of their patches is admitted exactly when each task's patches are admitted alone, and all interleavings produce the same state. Neither task waits for the other or is refused because of it.*

Appendix A gives the proofs. Like every optimistic scheme [42, 69], the guarantee reaches exactly as far as the read sets are sufficient and stable. A read made outside the store's reading interface is recorded nowhere, and over declared file sets an undeclared input can still hide a conflict (Section 16).

8 Liveness and attention

The empirical bottlenecks of Section 3 translate into eight operational invariants of the engine.

Deadlines, heartbeats and review windows. Every commitment carries a deadline t and a heartbeat interval h suited to its cadence: minutes for an analysis agent, a day for a site crew, a week for a thesis chapter. Only a `report` against φ renews h [5]. A lapsed heartbeat flags an active commitment *at risk*; an unpaused deadline expiry moves it to *violated* and escalates to its creditor. Region leases bind strictly to *active* commitments [53] and cannot outlive t without a creditor-accepted `amend`. When a punctual debtor submits a delivery `report` before t and α awaits the creditor's `decide`, entering *in review* pauses the debtor's violation clock for Δ_{rev} and places a timed review item on the creditor; if Δ_{rev} expires without a decision, the store escalates to the named arbiter (or applies contractual deemed-acceptance rules where specified).

Waiting is recorded, never polled. Blocked actors register waits explicitly, either as an antecedent ψ or as a watch on predicates of Π. When a committed delta touches a watched predicate, the store wakes the subscriber directly (Section 13). Agents do not poll, people do not chase updates, and the time work spends in queues, which dominates cycle time in product development [94], becomes a query that shows who is blocked on whom.

Lightweight commitments and anti-bureaucracy. Progress artifacts attach as Records via Attests; an explicit Commits assertion is created only where one actor genuinely relies on another. Early speech-act groupware such as The Coordinator failed when it forced workers to tag every conversational turn [106, 118]. Here, private checklists stay private, unsolicited drafts or observations enter freely as records before any promise exists, and an agent's internal plan imports as lightweight proposals requiring no human ceremony.

Asymmetric dependencies and symmetric co-design. When commitment c_1's declared write region $R_0(c_1)$ intersects c_2's observed guard region $R_{\text{obs}}(c_2, s)$ in one direction only ($R_0(c_1) \cap R_{\text{obs}}(c_2, s) \neq \emptyset$ while $R_0(c_2) \cap R_{\text{obs}}(c_1, s) = \emptyset$), c_1 produces inputs that c_2's acceptance reads [8, 21]; the store proposes discharged(c_1) as an antecedent of c_2 provided the antecedent graph remains acyclic. When the overlap is *symmetric*, as when the duct commitment c_M and the tray commitment c_E each run spatial clash checks across the same void, mutual antecedents would hold both in *conditional* for ever. A symmetric overlap is therefore reported as a *co-design coupling*, and the actors either order one before the other by `decide` or partition the zone.

Mode	Coordination pattern	Engine configuration	Examples
Run a playbook	known steps dispatched as prerequisites clear	workflow instantiates *achieve* commitments as antecedents discharge; exclusive R_0 leases; pre-authorised decisions; receipts as reports	wedding-day run sheet; site trade sequence; software release pipeline
Investigate	competing hypotheses; work spawned by new evidence	hypotheses are propositions linked to evidence via Attests; commitments test or gather evidence; read-shared regions; a decision adopts a hypothesis	scientific inquiry; failed slab inspection; production outage root-cause
Work together live	synchronous co-editing in a shared session	team holds one commitment over a shared R_0 under a CRDT/OT merge law with short h; decisions logged inline	co-authoring a paper; architectural charrette; human–agent plan sketching
Negotiate	offers, counter-offers and bilateral acceptance	offers are *proposed* commitments, counter-offers are amendments, contract formation is bilateral `commit` + `decide`	vendor quotes; subcontractor bidding; open task markets
Govern	formal decision rights, quorums, vetoes and appeals	`decide` checks named grants with quorum/veto predicates; `escalate` routes to a higher governance body	lab authorship rules; construction change orders; board resolutions
Maintain	continuous invariants spawning corrective work	standing commitments (κ = *maintain*, often $R_0 = \emptyset$); regression triggers repair commitments; handover via debtor `amend` (novation)	periodic lift/fire-door code checks; dataset schema health; p95 service SLOs

Table 2: Six collaboration modes as parameter configurations of one engine. Because all six read and write the same commitments, evidence and decisions, projects switch modes mid-flight without losing provenance.

Cost-of-delay attention ranking and root-cause coalescing. Because human attention carries high switching cost [60], decisions, violations, *at risk* warnings, review timeouts and refusal escalations merge into one priority queue scored by structural cost of delay [94]:

$$\mathrm{rank}(x) \;=\; \frac{\sum_{o \in \mathrm{outcomes}(x)} w(o) \cdot \big(1 + |\mathrm{blocked}(x,o)|\big)}{\max\big(t_{\mathrm{eff}}(x) - \mathrm{now},\ \tau_{\min}\big)},$$

weighting outcome priority $w(o)$ and the count (or resource value) of downstream conditional commitments blocked behind x against remaining slack $t_{\mathrm{eff}}(x) - \mathrm{now}$. Alerts that cascade along an antecedent chain are *coalesced to their root blocker*: if a level-4 concrete slab fails its strength test and holds twelve downstream trades in *conditional*, the contractor's queue shows one root item annotated with twelve blocked dependents.

Reliability measured by Percent Plan Complete. The fraction of commitments due in a window that were discharged on time, together with structured root-cause tags on every violation, gives Ballard's Percent Plan Complete [9] per team, actor and task category, giving planners empirical reference classes [45] without keystroke or time-tracking surveillance.

Creditor-owned acceptance and idempotent identities. Where α requires a creditor's `decide`, no debtor `report` can transition c to *discharged*, so an agent cannot grade its own work. Finally, every entity, proposal and external side-effect carries a store-issued idempotency key so retried network submissions never duplicate commitments or double-charge escrowed resources.

9 Modes, matching and markets

9.1 Modes as configurations of one engine

Groups coordinate in at least six distinct patterns, and a single project routinely shifts among them. Separate tools for each pattern break the project's history at every transition. The work graph expresses all six as configurations of four engine parameters: the *trigger* creating commitments, the *region policy* (exclusive R_0, merge-law shared R_0, or guard-only R_{obs}), the *decision rule* (authority grant, quorum or veto), and the *reporting cadence* (Table 2).

Because every mode mutates the same underlying assertions, transitions preserve full context. A building renovation executes as a playbook until concrete cylinder tests fail; shifts into investigation to isolate the weak bay; enters negotiation and governance to price and approve a structural change order; and resumes its playbook, all inside one continuous graph.

9.2 Matching work to actors

Routing by comparative advantage uses the existing ten verbs. A creditor (or coordinator acting under the creditor's grant) posts a *proposed* commitment with an open debtor slot or invites candidate actors. Candidates propose terms alongside verifiable credentials and trust-weighted capability histories (Section 5.3), and the creditor selects a debtor by `decide`, citing the competing proposals. This is Smith's Contract Net of announcement, bid and award [102], with signed credentials and records of past delivery behind the bids.

Projects encode routing policies directly in commitment templates: literature sweeps, boilerplate generation and rule checking route to bounded agents with human acceptance where stakes warrant; regulated sign-offs route strictly to

actors holding valid licences; and human escalations enter the cost-of-delay queue of Section 8. A solo maker operates identically, acting as root creditor of every outcome while delegating subtasks to agents and hiring occasional specialist reviews on demand.

9.3 Markets for complex work

An open market forms wherever open commitments meet qualified actors across organisational boundaries. Once a client selects a bidder, two coupled commitments form the bilateral contract: the worker commits to achieve φ by t (with review window Δ_{rev}), and the client commits to release payment with discharged(c_{work}) as its antecedent ψ. Escrowed funds sit as a Uses reservation on a money Resource, preventing double-spending, while multi-stage deliverables decompose via Contributes milestones. Timely delivery places c_{work} *in review*; if a strategic client stalls past Δ_{rev} without accepting or citing concrete defects, the liveness rules escalate the contract to the arbiter both parties granted, so the client cannot hold the escrow hostage.

Workers and clients both accumulate portable records, since a late or disputed payment by a client is itself the record of a commitment. Capability queries weighted by trust (Section 5.3) are designed to resist rating inflation [43] and Sybil wash-trading, and records that both sides hold reduce the dependence on one platform [97, 119].

9.4 Cross-organisation federation and ordering

Cross-organisational projects, from a university lab and a teaching hospital to an owner, general contractor and twenty subcontractors, rarely share one internal database. Each organisation retains its private subject space and internal commitments, exchanging only agreed cross-party commitments, attestations and decisions as signed records. To prevent split-brain transitions when two parties act concurrently on a shared commitment (for instance, a vendor submitting `cancel` while a client submits `amend`), every cross-organisation commitment designates an *authoritative host log* (or two-party hash-chained sequence) that totally orders state transitions on that commitment's identifier, while its acceptance query α is restricted to predicates over signed exchanged artifacts.

9.5 Domain neutrality

The engine standardises the mechanics of coordination and leaves the substance of each domain, such as building codes, the ethics of clinical trials, contract law or family norms, to domain vocabularies, verifiers and grants. Section 11 tests that separation across five disparate projects.

10 Subject spaces with structure

While the four status queries (Section 5.4) work over a plain folder of files, richer structure in the subject space sharpens the engine in five places: region precision, recorded read sets, identity-linked evidence, reactive watches and standing preconditions. We trace each using a building model and a software system, then specify online migration from flat files to a typed subject model.

10.1 Exact regions: fewer false refusals and missed conflicts

A commitment's lease $R_0(c)$ and guard $R_{\text{obs}}(c, s)$ are only as fine-grained as the predicates of Π_{sub}. When drawings sit as flat CAD or PDF files, the finest predicate is an entire sheet. Two mechanical engineers routing ducts in separate wings of level 3 touch the same sheet and collide under (A4) although nothing overlaps physically: a *false refusal*. Moving duct D-12 into the path of cable tray T-4 edits only the mechanical file; if the electrical clash check declared only the electrical and structural files, admission lets the edit pass and the collision surfaces on site: a *missed conflict* (Figure 3a). Over an IFC building model [62], R_0 names individual elements (`tray T-4`) while the spatial clash query dynamically populates R_{obs} with every geometric neighbour in the ceiling void (`duct D-12`, `beam B-7`), catching real clashes under (A5) while allowing safe clearances without locking unrelated trades (Figure 3b).

Software shows the same step. Over Git files, two agents editing adjacent functions in one file collide. Over a typed graph of the program and its design (Section 13.1), R_0 names a single function or contract field, and R_{obs} holds the resolved call and type edges that the check read.

10.2 Store-recorded read sets

When the subject store records every predicate read during a query, $Rd(\alpha, s)$, $Rd(\psi, s)$ and the read sets of checks are sufficient and stable by construction, which is what Theorem 1 assumes, and nobody has to trust build rules written by hand. Two intrinsic boundaries remain: *opaque interiors* (a pre-fabricated chiller unit or an unanalysed foreign binary is read as a single atom) and *external world lag* (queries over concrete cylinder strength or live production telemetry reflect the most recently imported observation record).

10.3 Identity-linked evidence and staleness

In a typed store, Attests links a Record directly to subject element identities at a specific revision [81]. When a subject element mutates in a subsequent revision, the engine distinguishes past completion from current validity: a *discharged*

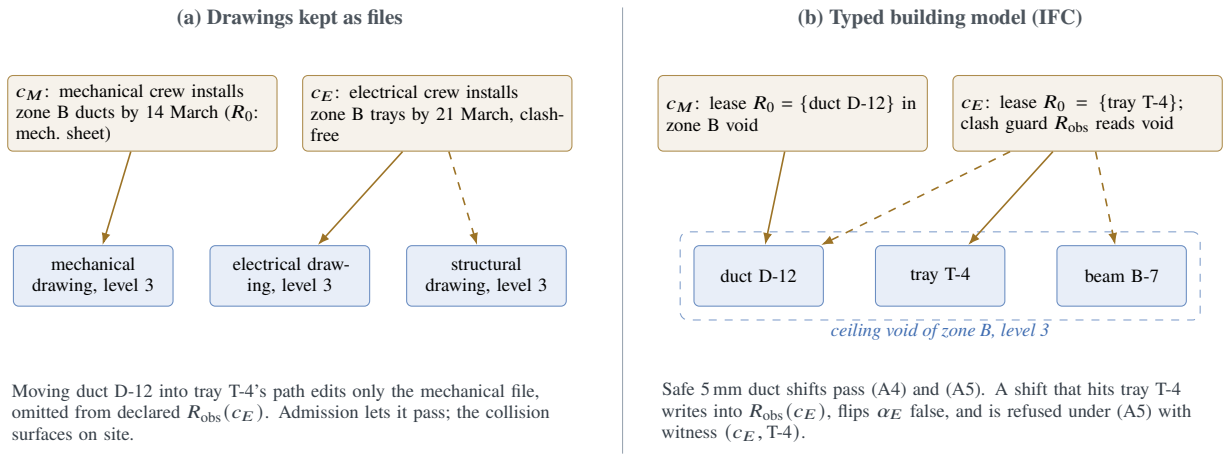

Figure 3: Lease regions R_0 (solid arrows) and observed guard regions R_{obs} (dashed arrows) over two subject spaces. With the two regions separate, c_M may adjust duct D-12 whenever clearance holds, and an edit that clashes is refused under (A5) with a witness before anything is fabricated.

one-off (*achieve*) commitment stays discharged in the history, and the proposition it established is flagged *stale* so downstream commitments depending on that proposition see that re-verification is needed (for example, when beam geometry changes after a structural stamp). An active *maintain* commitment is evaluated again at once under (A5): it refuses a change that breaks it, and it becomes *violated* when an external observation falsifies it.

10.4 Reactive watches and invariant preconditions

Because a structured store emits exact predicate deltas on every commit, recorded waits (Section 8) compile to zero-polling subscriptions woken only when watched extents change. Assumptions that span wide neighbourhoods also become enforceable, such as removing a partition wall provided it carries no load, dropping a dataset column provided no active pipeline reads it, or deleting an API field provided no deployed client calls it: the actor making the simplification records the global precondition as a standing *maintain* commitment ($R_0 = \emptyset$, $R_{obs} = Rd(\alpha, s)$). A later edit that routes load onto that wall or attaches a new reader to the dropped column writes into R_{obs}, since the query listed the column's readers, and is refused under (A5) with a witness naming the invariant's owner.

10.5 Online migration to a richer subject space

Projects often start over loose documents and move to structured models, from drawings to BIM, from notebooks to lineage tables, or from source files to a typed system graph, without changing kinds, families or verbs. Migration proceeds in five online steps:

1. *Preserve identities.* The target store retains every work-graph identity so commitments, grants, records and decisions keep unbroken provenance.
2. *Resolve subject references.* File-level predicates map to fine-grained target elements; ambiguous or unmapped references stay conservatively bound to their coarse declarations until confirmed by commitment holders.
3. *Recompute regions.* Active R_0 and R_{obs} sets are evaluated over the finer space, notifying holders whenever newly resolved read sets expose previously hidden overlaps.
4. *Rebind evidence.* Existing Records attach to resolved target elements so future edits automatically track staleness.
5. *Run a shadow period.* Both stores evaluate every change in parallel while the old space remains authoritative; disagreements measure the reductions in false refusals and missed conflicts (testing hypothesis **H3** in Section 15) prior to cutover.

11 Five projects through the engine

The engine is meant for any project with several actors, including one person directing many agents. Tables 3 and 4 compare five projects from different disciplines along the parts of the engine, and the subsections below note what each one needs most. The last subsection follows a building through a change of plan.

11.1 One maker with agents

One person building a model of a brain in a game engine is the creditor of every outcome and delegates most of the work. Each agent acts under the maker's grant, with a budget for computation and an expiry, so it cannot spend beyond what the maker allowed (Section 5.3). Agents record what they wait for, so the maker sees in one queue the few decisions that need a person, such as which atlas to trust or which simplification of the brain's connections is acceptable. Where the

Project	Actors	Subject space	Commitments
One maker with agents	the maker; agents that write code, build assets and search the literature; a neuroscientist and an artist hired for a few hours	the project's repository and assets; brain atlases as datasets	an agent imports the atlas regions as meshes by Friday; the neuroscientist reviews the model of connections
A research team	three students in three time zones, a mentor, a teaching assistant; agents for literature and analysis	a shared drive of documents and notebooks; versioned datasets	clean the data by Tuesday; draft the methods by Thursday; the mentor reviews the draft
A software startup	three founders; agents that write, review and test code; a contract designer	a repository, builds and the running service; a typed model of the system where one exists	a feature by a date; a release; a standing bound on a service's latency
A wedding	the couple, two families, a planner, vendors, an officiant; agents that compare quotes and track replies	a calendar of bookings, the guest list, the seating plan, the budget	the caterer serves 150 guests on the date; the venue holds the date; guests reply by a date
A building	owner, architect, licensed engineer, contractor, subcontractors, inspector, lender; agents for permits, clash checks and schedules	a building model, drawings, the schedule and contracts	each trade installs its systems in a zone by a date; the engineer stamps the drawings; the lender pays on an accepted milestone

Table 3: Five projects compared by their actors, subject spaces and commitments.

Project	Acceptance and credentials	Conflicts caught at admission	Outside parties and markets
One maker with agents	the maker's decision; the expert's review; a check on frame time	two agents editing one scene; a change that breaks the frame-time budget	experts and artists from a market, paid on acceptance
A research team	the mentor's decision; a check that the analysis reproduces from raw data; an ethics approval	two authors rewriting one section; an analysis of a dataset version that another member corrected	the data provider's terms; a conference's deadline
A software startup	tests and checks; a founder's decision on design; the app store's approval	two agents changing one module; a change that breaks what another commitment's tests read	a designer from a market; the app store's review
A wedding	the couple's decisions; the caterer's final count; the officiant's authority to register the marriage	two bookings for one evening; a guest-list change after the caterer's deadline; a deposit beyond the budget	vendors chosen from quotes; deposits held as resources
A building	inspections; clash checks; the engineer's stamp, which needs a licence	a duct and a tray in one ceiling void; a change to a stamped beam	subcontractors' bids; the inspector and the lender

Table 4: The same five projects compared by acceptance, conflicts and outside parties. Every row of both tables uses the same four kinds, six families and ten verbs.

model needs a neuroscientist's judgment, the maker posts an open commitment whose acceptance requires the expert's review, and pays on acceptance (Section 9.3). The maker never has to ask an agent for its status, since status is a query.

11.2 A research team

Members in three time zones are rarely awake at the same time, so promises with deadlines and heartbeats carry what a daily meeting would carry, and status is read by query. The mentor's acceptance query for a draft can combine a check that an agent can run, that the analysis reproduces from the raw data, with the mentor's own decision on the argument. Authorship and the sharing of data are decisions under grants, and a decision on authorship can cite the record of each member's contributions, as taxonomies of contributor roles propose [14]. Where datasets are kept with their lineage, a correction marks stale every analysis that read the old version, and the member who owns each analysis sees it.

11.3 A software startup

In a startup of three founders, most changes are proposed by agents working in parallel. Admission keeps them from colliding, and a refusal names the commitment and the actor to ask. Founders accept the work that needs their judgment, such as a design or a change of pricing, and agents cannot close it themselves. Standing commitments on the running service, such as a bound on the latency of an operation, turn a regression into new work in the mode of maintenance, and a release that needs an app store's approval waits on that outside decision as a recorded antecedent.

11.4 A wedding

A wedding has a fixed date, a budget, a dozen vendors from different organisations, and two families with views. Venues, rooms and the photographer's day are resources with capacities, so a second booking for one evening is refused at admission. The caterer's commitment has an acceptance query that compares the final guest count with the order, and its terms set a date for the final count. After that date, a change to the guest list that would make the query false needs an amendment that the caterer accepts. Guests' replies merge into the guest list under a merge law, deposits are money held

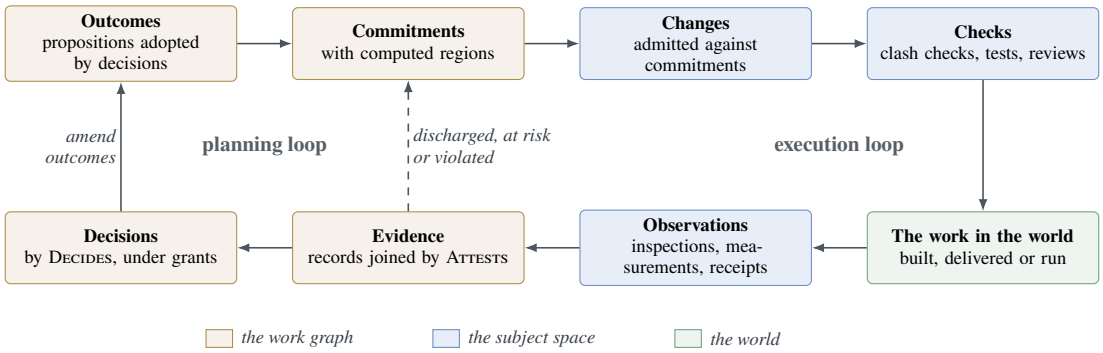

Figure 4: The two loops of a project. The execution loop runs from a commitment through admitted changes and checks to the work in the world, whose inspections, measurements and receipts become evidence and change the status of the commitment. The planning loop reads the same evidence, records decisions, and amends outcomes and commitments. The work graph holds the left half. The subject space and the world hold the right half: for a building, the model, its clash checks, the site and its inspections; for software, the repository, the builds and the running service.

as resources within the budget, and the officiant's authority to register the marriage is a credential that the ceremony's acceptance requires. Agents compare quotes and chase replies, and the couple decides, with each decision citing the quotes it considered.

11.5 A building through a change of plan

A playbook. The contractor runs the schedule as a playbook (Table 2). Each trade's commitment for a zone becomes active when its antecedent holds, such as the acceptance of the work it builds on, and agents keep the schedule by proposing the commitments that the plan calls for. The mechanical and electrical subcontractors hold active commitments over the ducts and the trays in the ceiling void of zone B on level 3 (Figure 3).

A refused change. The mechanical subcontractor's agent finds that duct D-12 passes under beam B-7 with insufficient clearance and proposes rerouting it. Because duct D-12 lies inside $R_0(c_M)$ and outside $R_0(c_E)$, clause (A4) permits the edit. D-12 also lies inside the observed guard region $R_{\text{obs}}(c_E)$ of the electrical clash check, which fails against tray T-4, so clause (A5) refuses the change with a witness naming c_E and the electrical subcontractor. The two subcontractors negotiate: the electrical subcontractor proposes lowering tray T-4 by 100 mm, amending c_E with the general contractor's consent, after which the duct move passes admission before anything is fabricated on site.

A failed inspection. The test cylinders from the slab of level 4 report a strength below the design value. The laboratory's report is a record attached by ATTESTS to the proposition that the slab meets its design strength, and the proposition fails. The concrete subcontractor's commitment is violated and escalated to the contractor, and every commitment whose antecedent needs the slab stays conditional, so the trades above it see what they wait for. The project moves into the mode of investigation. That the samples were mishandled, that the mix was wrong and that the slab cured in cold weather are hypotheses recorded as propositions, and cores drilled from the slab are evidence for or against each.

Decisions under grants. The cores confirm low strength in one bay. The structural engineer, whose licence the grant to decide structural remedies requires, decides to strengthen the bay and records the decision with the cores and calculations it considered. The engineer's earlier stamp on the drawings of that bay is marked stale until the revised drawings are stamped. The owner decides to accept the cost and the delay as a change order, citing the engineer's decision and the contractor's quote, and the decision amends the schedule's outcomes.

Work from a market. The contractor posts the strengthening as an open commitment whose terms require a certified installer of the strengthening system and whose acceptance needs the inspector's approval. Three firms propose, and the contractor decides, citing their proposals. The contractor's commitment to pay becomes unconditional when the inspector accepts the work.

Handover. When the building is handed over, the commitments of construction are discharged or released, and standing commitments of maintenance begin: the fire doors and the lifts are inspected at the intervals the local code sets, each inspection accepted by a qualified inspector. At every step, each of the four questions of Section 5.4 had an answer by query. Figure 4 draws the two loops that the project went around: the planning loop in the work graph, and the execution loop through the subject space and the site.

12 How the engine helps

Table 5 maps each problem of Section 2 to the mechanisms that address it, and to the evidence and the requirement of Section 3.9 behind them. The mechanisms address coordination. Some causes of failure lie outside it, such as optimism

Problem	Mechanism of the engine	Evidence and requirement
P1 Aims nobody can check	outcomes are propositions with queries, adopted by decisions; acceptance queries are agreed by creditors (Sections 5 and 6)	Last Planner; R1
P2 Promises nobody can see	commitments with debtor, creditor, deadline and heartbeat; who owes what to whom is a query (Section 6)	language action, Percent Plan Complete; R1
P3 Waiting nobody can see	waiting recorded as antecedents, review windows and predicate watches; queue time is a query (Section 8)	queues in product development; R2
P4 Status that has to be asked for	status computed by query; cost-of-delay queue with root-cause coalescing (Section 8)	surveys of work about work; R2
P5 Dependencies found late	asymmetric region overlaps proposed as acyclic antecedents; symmetric overlaps flagged for co-design (Section 8)	coordination requirements, clash detection; R3
P6 Changes that collide	split R_0 lease (A4) and R_{obs} non-regression guard (A5) with witnesses; CRDT/OT merge laws (Section 7)	clash detection, optimistic concurrency; R3
P7 Decisions without a record or a right	immutable Decides under grants, citing considered records; superseding chain (Section 5)	R7
P8 Evidence apart from claims	records linked by Attests; automatic staleness on subject mutation (Section 10)	building information models; R4
P9 Work closed by the actor who did it	creditor-owned α; review window Δ_{rev} protecting punctual debtors (Sections 6 and 8)	failures of agent verification; R4, R6
P10 Trust, credentials and liability	signed credentials in grants; mid-flight revocation; bounded agent scopes (Section 5.3)	verifiable credentials; R6, R7
P11 Shared capacity	threshold predicates $\pi_{r,k,I}$ and additive escrow deltas checked at admission (Sections 5 and 7)	coordination theory; R3
P12 Work across organisations	signed shared records sequenced per commitment by an authoritative host log (Section 9)	multi-university teams; R8
P13 Plans that are not kept	Percent Plan Complete with structured violation reasons; empirical reference classes (Section 8)	megaprojects, planning fallacy; R1
P14 Work given to the wrong actor	comparative-advantage routing on trust-weighted capability histories (Sections 5.3 and 9.2)	jagged frontier, human–AI pairs; R5
P15 Agents that coordinate badly	stated consequents and acceptance queries; obligations others can see; verification by another actor; grants and budgets; records read as data (Sections 6 and 8)	failure taxonomies and benchmarks of agents; R6

Table 5: Each problem of Section 2, the mechanisms that address it, and the evidence of Section 3 and requirement of Section 3.9 behind them.

in estimates or a lack of skill, and for those the engine shows the gap early and leaves the remedy to the actors.

For people. People spend less of the day asking for status, reporting it and aligning it, because status is computed and everyone reads the same answer. What needs a person reaches them in one queue, ranked by the outcomes it threatens, and each request for a decision arrives with the records it should consider. A person's private list stays private, and commitments are asked for only where another actor depends on them.

For agents. An agent gets what the evidence of Section 3.7 says it lacks: a stated consequent and acceptance query, obligations of other actors that it can read, a grant and a budget that bound what it may do, a way to wait without polling, and a refusal that names whom to ask. Its work is accepted by its creditor, so an error in its own judgment of completion is caught before others build on it.

For experts and certified actors. An expert's time goes to the acts that need the expert: a judgment, a review, a stamp, an approval. Agents and other actors prepare the evidence, the acceptance query states what the expert must decide, and the decision is recorded under a grant that names the credential it required. The record shows who decided what and on which evidence, which is what liability and audit need.

For organisations and markets. Organisations exchange commitments and decisions as signed records, so each party sees what it was promised and by whom, while keeping its own tools. In a market, the same records are the offer, the contract, the progress, the acceptance and the reputation, and both sides hold them (Section 9.3).

What the engine leaves to the actors. The engine does not make estimates better, and it does not give actors skills they lack. It shows early when promises slip, and it keeps a history of commitments that a planner can use as a reference class for new estimates, the method that Flyvbjerg recommends against optimism in planning [45]. Domain standards, the content of the work and its quality stay with the actors and the checks they choose.

13 One graph model and concurrent engine

Two separate databases, one for the work graph and one for a typed subject model such as Reaktor's system graph, would duplicate identities, patches, read-set tracking, watches and admission. A coordination graph kept as objects with pointers and mutexes does not serve many agents at once either: locks per node deadlock across patches that touch several elements, and a compare-and-swap per edge cannot evaluate an acceptance query over several predicates (A5) on one

consistent snapshot. This section addresses both problems. First, it extends the port-hypergraph schema and meta-model of typed system models so that coordination and structured subject models are instances of *one* mathematical graph model. Second, it designs a concurrent persistent hypergraph store in which readers pin snapshots without locks, checks are evaluated in parallel, only a short commit step is serialised, and heartbeats never write to the graph. Third, it reports two implementations: the concurrent graph in C++23 and the whole engine in Kotlin (Section 13.3).

13.1 One graph model for coordination and subject structure

The typed port hypergraph used here comes from the Reaktor Graph design study [100]. The study models a graph as a functor from a schema category $\mathcal{S}_{\mathsf{RG}}$ to sets, whose objects are nodes, ports, hyperedges, three objects of incidences split by the sort of their end, and containment links. The study was an exploration of the design and will not be implemented as written. Reaktor moves from the graph that runs today directly to its system graph, a later model that keeps the study's schema, is typed through three tiers (a fixed meta-model T_{meta}, imported domain vocabularies $V \to T_{\text{meta}}$, and system instances $G \to V$) and has the kinds, families and modalities of Definition 4. Because the schema lets an incidence end on a node, a port or another hyperedge, the work graph can use assertions about assertions directly: Uses attaches a reservation to a Commits hyperedge, Commits and Decides cite a Grants hyperedge as `authority`, and Decides supersedes an earlier Decides. Relative to the study, one model for work and for the structure of a subject needs four additions.

1. *Entities treated alike in footprints.* Admission needs to know exactly what a change reads and writes. The work graph gives every entity, whichever its sort, the same three kinds of predicate of Definition 1: existence, extent and the family of its dependents. Read sets, write sets, leases and guards are then stated once for nodes, ports and hyperedges alike, and a commitment's region can name an assertion as easily as an element.
2. *Coordination families.* The study's Resource kind names a provider's resource, such as a bucket or a queue, and its Grant family carries budgets. It has no Commits, Decides, Contributes or Uses, so promises, decisions, decomposition and reservations of shared capacity (hours, money, equipment) lie outside it.
3. *Levels chosen by each domain.* The study fixes six layers for software: source, definition, deployment, runtime, observation and assurance. Buildings, research datasets and curricula have levels of their own. The unified model lets each vocabulary bring its own chain of levels, all starting at Intent.
4. *Bilateral commitments.* A system model checks invariants at commit, each with one owner. It has no exclusive write lease for a debtor (A4), no reservation of resources (A3) and no timed lifecycle.

Definition 4 (Schema $\mathcal{S}^{+}_{\mathsf{RG}}$, graphs and the unified meta-model T^{+}_{meta})**.** Let $\mathcal{S}^{+}_{\mathsf{RG}}$ be the category with objects N, P, E (nodes, ports and edges), I_N, I_P, I_E (incidences that end on a node, a port or an edge), C (containment links) and D (scope declarations), generated by the arrows

$$\begin{gathered}\text{own} : P \to N, \qquad \text{edge}_X : I_X \to E, \qquad \text{end}_X : I_X \to X \quad (X \in \{N, P, E\}),\\ \text{child}, \text{parent} : C \to N, \qquad \text{declared} : D \to E, \qquad \text{in} : D \to N.\end{gathered}$$

A *graph* is a functor $G : \mathcal{S}^{+}_{\mathsf{RG}} \to \mathsf{Set}$ with typed attributes, among them a role and a position on every incidence and a kind or family, a modality, a level and an extent on every entity, added as in attributed C-sets [85]. Graphs and their natural transformations form the category $\mathsf{URG} = \mathsf{Set}^{\mathcal{S}^{+}_{\mathsf{RG}}}$. The *entities* of G are $\text{Ent}(G) = G(N) \uplus G(P) \uplus G(E)$. A graph is *well formed* when every node is the child of at most one containment link, the containment links form no cycle, every edge has at most one scope declaration, no two incidences of one edge have the same role and position, and the typing rules of its vocabulary hold. The unified meta-model $T^{+}_{\text{meta}} \in \mathsf{URG}$ has:

- **8 meta-kinds** (Table 6): the 4 coordination kinds (Actor, Proposition, Record, Resource) and the 4 structural kinds of typed system models (Scope, Operator, State, Type);
- **15 meta-families**: the 6 coordination assertion families (Commits, Attests, Decides, Grants, Contributes, Uses), the 4 operational connection families (Bind, Session, Channel, Transition), and the 5 structural assertion families (Export, Access, References, Derives, Refines);
- **4 modalities** {intended, realised, observed, claimed}, and for each vocabulary a chain of levels $(\Lambda, \leq)$ whose least element is Intent.

The schema is the study's with one more object, D, for the scope of an edge, which the study states in its definition of a graph but not in its schema. Partial maps such as the parent of a node and the scope of an edge are written as spans through C and D, so every arrow of the schema is a total function in every graph. The conditions of well-formedness are conditions on objects of URG and are not part of the category.

Proposition 4 (The ambient category)**.** URG *is a presheaf topos, and so is its slice* URG/T *over any object* T*. Both are adhesive categories.*

Appendix A gives the proof. Adhesiveness [70, 85] is what lets patches be written as double-pushout rules with the standard theory behind them, including the Local Church–Rosser theorem for parallel independent rules. A morphism of

Sub-signature of T^{+}_{meta}	Kinds and families	Role across disciplines
Coordination core Σ_{work}	**Kinds (4):** Actor, Proposition, Record, Resource **Families (6):** Commits, Attests, Decides, Grants, Contributes, Uses	Present in every project at Intent and across modalities: outcomes, promises, escrow budgets, credentials, evidence and governance
Subject structure Σ_{sys}	**Kinds (4):** Scope, Operator, State, Type **Connections (4):** Bind, Session, Channel, Transition **Assertions (5):** Export, Access, References, Derives, Refines	Present when the subject space is a typed model in the store: Reaktor software levels, IFC building components, lineage pipelines or Manna's knowledge graph

Table 6: The unified meta-model T^{+}_{meta} (8 kinds, 15 families) splits into the coordination core Σ_{work} of Section 5, present in every project, and the structural complement Σ_{sys}, both stored in one C-set hypergraph.

vocabularies f induces the functorial data migrations Δ_f, Σ_f and Π_f between the slices [103], which is how the shadow migration of Section 10.5 moves instances. One caution applies throughout. Because well-formedness is not part of the category, a rewrite that is valid in URG can produce a graph that is not well formed. The paper never relies on the category to prevent that: clause (A1) checks it for every patch, and the reads of that check are part of the patch's read set (Section 7).

The four modalities say how an entity is known. An *intended* entity says what should be: a requirement, a plan item, a design. A *realised* entity exists because an admitted action made it so and returned a receipt. An *observed* entity records what was read from the world at a time, such as an inspection or a measurement. A *claimed* entity is asserted by a person, an agent or a model, with or without evidence. As a design that neither implementation carries yet, every realised, observed or claimed entity also has a status vector computed from its evidence: method, freshness, coverage and agreement. The staleness of Section 10 is the freshness component.

Levels are data of a vocabulary. A software vocabulary may use, for example, the chain Intent < Design < Program < IR < Artifact < Place < Machine, which is the example software vocabulary of the Kotlin engine. A construction vocabulary may use Intent < Spatial < System < Element < Site, a curriculum Intent < Domain < Resource < Chapter, and flat files Intent < Subject. An invariant that a system model checks at commit becomes a guard in the sense of Definition 2: a standing commitment $(d = a, r = a, R_0 = \emptyset, \kappa = \mathit{maintain}, t = \infty)$ of its owner a, whose acceptance query is the check of the invariant. Invariants of software and promises between people and agents then pass the same gate (Definition 3).

13.2 Architecture of the concurrent hypergraph engine

Figure 5 shows the architecture of the store. It combines five techniques from persistent data structures, database concurrency control and high-speed networking:

1. **Persistent 4,096-row columnar chunks (MVCC).** Following data-oriented design [41] and path-copying persistence [30, 33], each C-set sort is stored as parallel arrays partitioned into immutable chunks of $2^{12} = 4{,}096$ rows. A `Snapshot` holds `shared_ptr` chunk tables and a monotonically increasing `RevisionId` [12]. A reader pins the active snapshot without locks or retries: it announces the current epoch in its own cache-line slot and loads the root pointer, and the writer frees a retired root only once every reader has announced a later epoch (epoch-based reclamation [47]). An atomic `shared_ptr` would be simpler, but standard libraries implement it with an internal lock, so pinning would no longer be wait-free. Mutating m rows copies each touched column's chunk table, one pointer per 4,096 rows, and at most m chunks, leaving historical snapshots intact. Nodes, ports and edges each have an entity table. An edge's incidence rows sit together in one immutable list, an incidence may end on another edge, and every entity keeps the list of entities that depend on it: its ports, its children, the edges that end on it and the edges scoped to it.
2. **Automatic bitset read-set capture.** Every traversal, check and acceptance query reads columns exclusively through a `TrackedReader` wrapper that sets the corresponding predicate bit in a thread-local `PredicateSet` (`uint64_t` words). Recorded read sets $Rd(q, s)$ are therefore sufficient and stable by construction for every read made through the wrapper. A read set over 10^5 predicates occupies 12.5 KB, which fits in a core's L1 data cache. Each snapshot also keeps, for every predicate, the revision that last wrote it, so testing $Rd(P) \cap Wr(\Delta_{(v,v']}) = \emptyset$ reduces to checking that no predicate in $Rd(P)$ was written after v: one lookup per predicate read, however many changes were admitted since v, as in per-record validation [110]. Every entity has three predicate identifiers: its existence, its extent and its dependents. The third stands for the whole family of membership predicates of Definition 1, with one last-write stamp for the family. Attaching an entity sets that stamp and reads nothing of it; listing the dependents reads it. Because the commit step compares reads with writes and never writes with writes, two attachments to one entity do not conflict, which is exactly what the family gives, and a listing conflicts with any later attachment, which rules out phantoms. Following an edge reads only the existence of its endpoints, so it does not conflict with a change to an endpoint's extent.

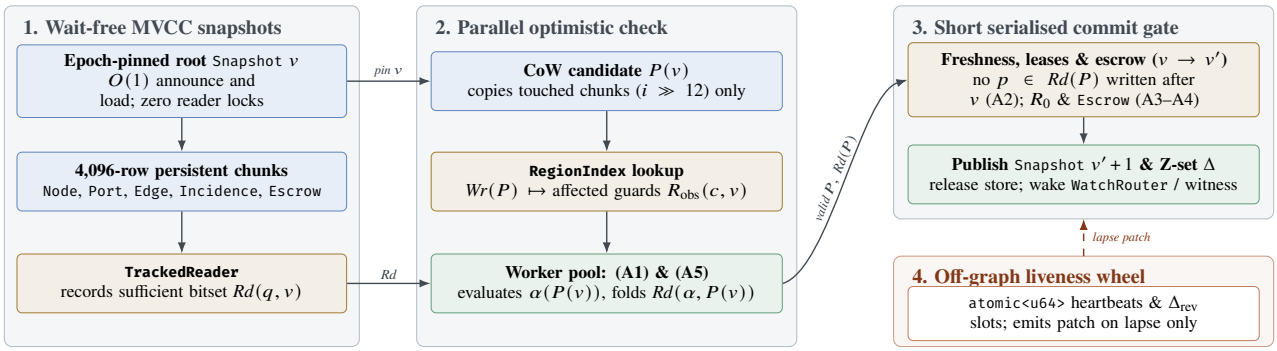

Figure 5: Architecture of the unified concurrent hypergraph store. Readers and check workers pin immutable snapshot roots in $O(1)$ without locks. Candidate changes run expensive (A1) and (A5) queries in parallel off the lock, recording post-state bitsets via `TrackedReader`. The serialised commit gate evaluates no queries: it checks each read predicate's last-write revision, escrow capacity and R_0 leases, records reservations and swaps chunk pointers, which is designed to take microseconds (a target that has not been measured), while an off-graph timing wheel absorbs high-frequency heartbeats without copying column chunks.

Store operation	Sync. class	Time / space bound	Mechanism
Pin snapshot for query / check	Wait-free	$O(1)$ time, 0 allocs	Announce the epoch, then load the immutable `Snapshot` root
Read entity / CSR incidences	Lock-free	$O(\deg) + 1$ bit op	Chunk shift `slot >> 12`, bitset mark in `TrackedReader`
Validate (A1) & affected (A5)	Lock-free ‖-ism	$O(\sum_{c \in \text{hit}} \lvert\alpha_c\rvert)$	Parallel over CoW overlay; skips untouched commitments
Commit gate rebase ($v \to v'$)	Short mutex	$O(\lvert\Pi\rvert/64 + \lvert Rd(P)\rvert + \lvert Wr(P)\rvert)$	Last-write revision per read predicate, $O(1)$ R_0 & escrow check, chunk-table copies
Wake blocked actors (watches)	Lock-free fan-out	$O(\lvert Wr(P)\rvert \cdot \text{subs})$	Z-set delta $\Delta_{v'+1}$ routed through inverted watch index
Renew heartbeat on `report`	Wait-free	$O(1)$ time, 0 chunks	Relaxed/release atomic store in off-graph `LivenessWheel`

Table 7: Concurrency guarantees and asymptotic bounds of the unified store for $|\Pi|$ active predicates, read set $Rd(P)$ and patch footprint $|Wr(P)|$. The bounds are asymptotic; no timings have been measured.

3. **Two-tier inverted region index and additive escrow.** Each snapshot maintains a `RegionIndex` mapping every predicate π to (a) its single active R_0 lease holder (if any) for $O(1)$ clause (A4) checks, and (b) the compact list of commitments whose $R_{\text{obs}}(c, v)$ contains π, so clause (A5) re-evaluates *only* commitments whose observed predicates were actually touched by $Wr(P)$. Shared Resource capacities sit in an `EscrowLedger`. A reservation is tested inside the commit step against what is free at the head, $\text{avail}(r, v') \geq k$, and records no read of the resource's row, so reservations from one budget do not abort one another under (A2). The index and the ledger are stored in chunked columns like the rest of the snapshot, so a change copies only the chunks it modifies.
4. **Two-phase parallel optimistic admission.** In *Phase 1*, with no lock held, a submitting worker pins snapshot v, builds a candidate `Snapshot` that shares unmodified chunks, and runs the expensive (A1) checks and the affected (A5) acceptance queries in parallel, folding their read sets before and after the change into $Rd(P)$. In *Phase 2* (serialised), the commit gate checks that no predicate in $Rd(P)$ was written after v [42] and that no commitment began guarding $Wr(P)$ after v, verifies grants and escrow capacity at v' (A3) and R_0 leases (A4), records the reservations and any at-risk marks, rebases the modified rows onto v' if $v' > v$, atomically publishes $v' + 1$, and pushes the Z-set delta $\Delta_{v'+1}$ [17] to wake registered predicate watches. Phase 2 evaluates no queries, so it is designed to take microseconds. That figure is a target: the implementations of Section 13.3 are tested for correctness, and neither has been timed. By Proposition 2, evaluating the queries in Phase 1 at the base and repeating only the freshness test at the head gives the verdict that admission at the head would give. Patches obey the dangling condition of double-pushout rewriting: an entity is retired only together with every entity that still depends on it. A retired slot is reused only under its next generation, so a stale handle fails validation (A1) and never reaches the slot's new occupant.
5. **Off-graph atomic liveness wheel.** Frequent `report` heartbeats must not allocate persistent graph chunks on every renewal. An in-memory `LivenessWheel` [116] holds one cache-aligned atomic timestamp slot per active commitment: `report` moves the slot's `next_hb_due_ns` with a single `memory_order_release` store, while a background ticker emits a graph patch *only* when a heartbeat lapses, a review window Δ_{rev} expires, or a deadline passes.

13.3 Reference implementations

The design is implemented twice, and both implementations are reproduced in full in the appendices, read from the same files that their tests compile.

The concurrent graph in C++23. Appendix D gives the concurrent hypergraph store of Section 13.2 in five headers and about 870 lines. Nodes, ports and edges live in per-sort entity tables addressed by 64-bit handles that pack a sort tag, a generation and a slot (Listing 2). A graph patch creates and retires entities, sets extents and installs commitments with their regions (Listing 4). The admission gate pins a snapshot, validates the patch, runs checks and affected guards on a candidate snapshot, and then, under a short lock, refuses on stale reads, on guards that formed after the pin, on grants and escrow, and on leases, before it rebases the patch onto the head and publishes it (Listing 5). The 40 checks of its test program pass in an optimised build and under the address, undefined-behaviour and thread sanitizers; the code dates from 12 November 2026, and the three builds were run again on 16 November 2026 with all 40 checks passing in each. The checks include four writers adding 1,600 edges onto one hub without a refusal while two readers verify every snapshot they pin, and a read-modify-write contended by four threads that loses no increment (Listing 7).

The whole engine in Kotlin. Appendix E gives a Kotlin engine for the whole specification in eleven files and about 1,330 lines. It covers the unified model, persistent columns and read sets, the commitment tuple and its states, snapshots with their region index, escrow holds and delegation index, the five-clause admission gate with witnesses (Listing 14), the ten verbs, liveness and the reactor, the status queries with root-cause coalescing, Percent Plan Complete and trust-weighted capability, modes, matching, escrow-backed contracts and the federation host log, and subject spaces with shadow migration. It is written for clarity, and its speed has not been measured. A snapshot is an immutable value made of chunked copy-on-write columns. Admission builds and checks a change on a pinned snapshot, and the commit step is a compare-and-set loop on one atomic reference, so a writer never waits for a lock, and the garbage collector reclaims every snapshot that no reader still holds. Its 59 tests, which all passed in the test run of 12 November 2026, follow the paper's scenarios, among them the ceiling-void clash of Figure 3, fifty agents reserving one budget at once, the late slab whose twelve blocked trades surface as one item, a self-dealt history that earns no capability, and a shadow migration that reports both a false refusal and a missed conflict.

What the two share and where they differ. Both implement the same five clauses, return the same witnesses and keep a region index. Both record every read made through their reading interface, `TrackedReader` in C++ and `View` in Kotlin, so those read sets are sufficient and stable by construction; a read made outside the interface is not recorded. Both test reservations at the head inside the commit step, which is the rule that only reservations read threshold predicates. They differ in purpose. The C++ store models the graph exactly and is built for concurrency. It leaves the meaning of grants, leases and propositions to a policy supplied by its host, and it has no verbs or queries of its own. The Kotlin engine implements every verb, query and protocol in the paper. It keeps one predicate per element, has no ports and never deletes an element, so the dangling condition does not arise in it. Neither has been timed.

Where the code is narrower than the theorem. Three differences between the two implementations and the assumptions of Theorem 1 remain, and none of them is fixed in the code of 12 November 2026. First, the theorem treats a patch as reading what it replaces. Both stores record that read when the change reads the value before replacing it, as the read-modify-write tests of both do. A change that replaces an extent without reading it records no read, so two such blind writes to one extent from one base are both admitted and the later one is kept. Such pairs fall outside the theorem, and the history is still serialisable, because neither change depended on the value it overwrote. Second, the theorem counts the reads of a new commitment's queries in the read set of the change that creates it. Neither store adds them. Both refuse at the commit step a change whose writes meet a region formed after its base, which covers changes committed after the commitment; a change committed between the base and the commit of the creating change can still leave the new commitment's recorded region out of date. Third, the C++ store keeps the observed region that its host supplies when a commitment is installed and does not record it again after evaluating the guard, while the Kotlin engine records it again on every complete evaluation.

14 The engine in Hangar and Manna

The specification of Sections 5 to 9 and the store of Section 13 do not depend on any one product, and Appendix B maps existing kinds of tool onto its primitives. Two products are meant to run on the engine: Reaktor's Hangar for software systems, and Manna for projects across disciplines, structured learning and an open market for work. As of 16 November 2026 neither does: nothing in either product imports the C++ store or the Kotlin engine, and everything in this section beyond the present state of the two products is a design.

Reaktor and Hangar. Reaktor is a multiplatform framework for phones, desktops, browsers and servers [92]. Its desktop workbench, Hangar, coordinates people and AI agents through a single catalogue of operations [90, 91]. Today Hangar keeps each plan as a `WorkPlan` whose items are components, tasks, checks and decisions, with statuses, dependencies, the seat that works on an item and a time-bounded `PlanLease`; these hold part of the state of a commitment under other names. The design mounts Hangar on the store of Section 13, first over repositories, builds and code-graph providers, and later beside the typed Σ_{sys} vocabularies of Reaktor's system graph (Section 13.1), through the shadow migration of Section 10.5 [93]. Reaktor goes from the graph that runs today directly to that model. The earlier Reaktor Graph design study [100] is not a step on the way and will not be implemented. Hangar is written in Kotlin, so the Kotlin engine of Appendix E could run inside it before the C++ store is mounted.

Hypothesis	Measure	Rejected if
H1 Commitments with deadlines, heartbeats and creditor acceptance raise promise reliability.	Weekly Percent Plan Complete by debtor class on size-matched commitments before vs. after adoption.	PPC fails to rise within eight weeks, or rises solely via commitment shrinkage.
H2 The unified core Σ_{work} (4 kinds, 6 families, 10 verbs) expresses coordination across disciplines without opaque workarounds.	Share of coordination events in messages, minutes and issue logs represented structurally without new primitives.	any recurring coordination event needs a new kind, family or verb, or hides coordination state inside unqueried Record text.
H3 Typed subject spaces and split (R_0, R_{obs}) regions cut both false refusals and missed conflicts relative to files.	Disagreements logged during shadow migration (Section 10.5), adjudicated by commitment holders.	moving to the typed space fails to reduce both error classes.
H4 Protocol bookkeeping costs less human time than the coordination overhead it eliminates.	Experience-sampling time use before vs. after; fraction of commitments drafted by agents; off-protocol work rate.	human coordination time rises, or actors systematically bypass the store.
H5 Queryable status and root-cause-coalesced cost-of-delay queues cut both status polling and decision latency.	Weekly status-inquiry messages and median decision queue wait time.	*either* metric fails to fall (or increases).
H6 Comparative-advantage routing on trust-weighted capability evidence beats habitual assignment.	Randomised assignment comparing first-pass creditor acceptance, rework and cost per discharged commitment.	evidence-based routing fails to improve cost-adjusted acceptance.
H7 Trust-weighted commitment histories predict on-time accepted delivery better than star ratings.	Out-of-sample calibration predicting timely discharge in an open market.	predictive accuracy does not exceed star ratings.
H8 Persistent MVCC chunks, last-write freshness checks and threshold escrow sustain parallel multi-agent throughput.	Commit throughput, p99 gate hold time and spurious-(A2) abort rate with 32–128 concurrent agent workers.	commit gate p99 exceeds 50 μs or shared-budget proposals suffer serial hot-spot aborts.

Table 8: Eight falsifiable hypotheses, empirical or benchmark measures, and strict rejection criteria.

Manna. Manna organises projects, research drives, roadmaps and a typed prerequisite graph of books, courses and chapters [77]. Today Manna stores each plan as one record that holds the whole `WorkPlan` as a JSON string, so plan items are not entities with identities of their own. In the design, Manna runs on the same store under its own vocabulary of levels. Learning completions and project deliverables become evidence of capability, open commitments match tasks to people, agents and certified specialists, and payment held in escrow and released on acceptance gives the market of Section 9.3.

One graph across both products. If both products share T^{+}_{meta} and one store, a project can span them. A neuroscience team that coordinates its literature review and ethics approval in Manna could delegate simulation code to coding agents in Hangar, while outcomes, grants, evidence and decisions stay in one graph. This is a design and has not been built.

15 Hypotheses and evaluation

Every central claim of the engine is falsifiable. Table 8 states eight hypotheses across human–agent teams, cross-disciplinary projects and the concurrent store, together with operational metrics and explicit rejection criteria.

Staged adoption and baselines. Adoption decomposes into four independently valuable stages: Stage 1 deploys commitments, heartbeats, recorded waits and status queries over existing drives and chats (testing **H1**, **H4**, **H5**); Stage 2 enables admission with grants, credentials and resource escrow over declared regions (testing **H2**, **H8**); Stage 3 migrates onto a typed subject model via shadow evaluation (testing **H3**); and Stage 4 opens commitments to escrow-backed markets (testing **H6**, **H7**). Pre-adoption baselines reconstruct historical Percent Plan Complete, status-request rates and decision latencies from archived tickets and chat logs across size-matched work items. Because heartbeats respond solely to deliverable `report` actions, evaluation never inspects keystrokes, screen time or physical location.

16 Risks

The mechanisms of this paper reduce the ways the engine can fail, and some risks remain. We order the remaining risks by how severely each could undermine adoption or fairness.

Commitments can ossify into bureaucracy. Requiring explicit promises for trivial edits causes people and agents to route work around the store [56, 106, 118]. Keeping private lists private, recording explicit commitments only at genuine inter-actor dependencies, importing agent sub-plans automatically and letting agents draft routine bookkeeping mitigate friction; hypothesis **H4** tests whether that suffices.

Work records can be repurposed for surveillance. Algorithmic management systems frequently slide from coordination into coercive monitoring [66]. By design, heartbeats renew strictly through deliverable reports at mutually agreed

intervals, private notes stay off-graph, and parties hold bilateral copies of every record. Because a hostile operator could still weaken grant policies, self-custodied signed records remain essential.

Algorithmic routing and reputation can entrench bias. Opaque platform allocation harms worker autonomy [72, 97, 119] and falls under strict statutory oversight in the EU Platform Work Directive and AI Act [39, 40]. Making routing rules inspectable graph propositions, keeping account-ending decisions with authorised humans, and anchoring disputes in mutually granted arbiters align the protocol with transparency mandates, though deployments must still audit domain-specific compliance.

Acceptance queries can be Goodharted. Any automated check pressured as a target invites gaming [105]. Acceptance queries stay with the creditor, and a `decide` by a person or a licensed expert is required wherever judgment resists formalisation, so an agent cannot pass by exploiting a brittle test suite.

Reputation inflation and Sybil wash-trading. Marketplace ratings drift upward over time [43], while open networks invite Sybil self-dealing [23]. Weighting capability histories relative to the viewer's trust roots, scaling by commitment value and human-judged acceptance, and exposing full outcome distributions alongside summary scores harden reputation; hypothesis **H7** measures what signal remains.

Undeclared read sets and unmodelled physical couplings. Over flat files without hermetic sandboxes [108], an omitted file declaration makes a read set insufficient, and Theorem 1 then no longer applies, so the store reports the share of admissions that rest on declared read sets. Even in a typed model, a physical interaction that no standing commitment checks, such as the cumulative load on a floor when nobody registered an invariant for the slab, stays invisible until someone states the invariant or observes the world.

Attention overload. Merging violations, risks, refusals and decisions into one queue risks alert fatigue if cascading failures fan out unchecked. Ranking by cost of delay and coalescing along antecedent chains (Section 8) collapse downstream cascades into single root blockers; how to tune them with many agents at once is open (**H5**).

Prompt injection, legal limits and leaks across organisations. Untrusted external documents attached as Records can carry instructions aimed at the agents that read them [54]; keeping records apart from instructions, narrow grants and fixed caps on escrow limit the damage. Externally, signed records supply evidentiary provenance without replacing statutory contract or tort law, and cross-organisation federation shares only explicitly exported commitments so internal decomposition stays confidential.

17 Related work

Linguistic action and commitment protocols. Winograd and Flores framed organisations as conversations for action [117, 118], inspiring ActionWorkflow [78] and DEMO [32], while Suchman critiqued rigid speech-act categorisation [106]. In multi-agent systems, Singh grounded interaction in social commitments [101], Yolum and Singh compiled protocols as event-calculus commitment machines [125], and Castelfranchi, Jennings, Cohen, Levesque, Grosz, Kraus, Tambe and Rao developed joint intentions, SharedPlans and BDI execution [20, 26, 27, 55, 64, 89, 107]. The work graph adds to commitment machines a computed region (R_0, R_{obs}) for every commitment over a versioned subject store, and it enforces the promises when a change is admitted.

Coordination, scheduling and lean construction. Malone and Crowston classified resource, prerequisite and decomposition dependencies [75]; Smith's Contract Net formalised task bidding [102]; and blackboard/tuple-space architectures such as Hearsay-II and Linda coordinated specialists over shared state without ownership leases or creditor acceptance [37, 49, 57]. Critical Path [65], Critical Chain [51], Scrum [98], Kanban [3] and Ballard's Last Planner System [9, 73] schedule or track promises at the plan level, while ISO 19650, IFC and BCF attach issues and approvals to building elements [19, 62, 63]. The work graph joins the promise tracking of the Last Planner System with admission against spatial and structural checks before conflicting work is built.

Categorical graphs, concurrent stores and incremental views. Typed port hypergraphs and C-set databases build on adhesive categories, double-pushout rewriting and functorial data migration [34, 46, 70, 85, 103]. On the systems side, workflow nets, BPMN, sagas, leases and level-triggered controllers handle process choreography and fault recovery [48, 53, 83, 109, 113, 114]. Our concurrent store combines Driscoll's path-copying persistence [33], used in functional graph stores such as Aspen [30] and in snapshot streaming systems such as Kineograph [24], with multiversion concurrency control [12], Kung–Robinson optimistic validation [69], the read tracking of serialisable snapshot isolation [42], predicate reads that rule out phantoms [38], escrow for shared quantities [84], sparse CSR incidence layouts [67], DBSP Z-set delta streams [17], CRDT/OT merge laws [35, 99] and hashed timing wheels [116].

Agent protocols, crowd markets and provenance. Contemporary multi-agent frameworks orchestrate turns or static DAGs [31, 59, 71, 88, 121]; MCP standardises tool invocation [4] and A2A tracks point-to-point task states [1], and neither provides shared artifact regions, admission against other promises or escrowed markets. Flash organisations [96, 112], reputation systems [95], W3C PROV [81] and Verifiable Credentials [104] supply institutional building blocks that the work graph brings into one engine.

18 Conclusion

Complex projects in every discipline now bring together people, autonomous agents, contractors and licensed experts whose unrecorded promises, invisible queues and late-discovered collisions consume much of the working day. The work graph addresses those failures with one small coordination core of four element kinds, six addressable assertion families and ten lifecycle verbs, which turns aims, obligations, evidence that changed the plan and the right to decide into graph queries. Splitting a commitment's declared write lease R_0 from its read-tracked guard region R_{obs}, requiring the consent of both parties and creditor review windows, and testing reservations against what is free let actors and agents whose changes are independent work without waiting for one another, and they surface semantic collisions early with explicit witnesses. Independence has a precise meaning: neither change sets a predicate the other reads, with the dependents of an entity read as a family and quantities read only by reservations.

Extending the meta-model of typed system models with the six coordination families, the same three kinds of predicate for every entity and a chain of levels per domain puts the coordination of people and agents and structured system models in one adhesive category of presheaves and in one persistent chunked multiversion store, which is implemented and tested and not yet deployed. The engine is implemented whole in Kotlin and its concurrent graph in C++23, and both are reproduced with their tests in the appendices. Projects can adopt the protocol step by step, starting over ordinary folders and chats and migrating later onto typed models of buildings, datasets or software, while carrying one unbroken record across teams, organisations and open markets for complex work.

References

[1] A2A Project. Agent2Agent (A2A) protocol specification. https://a2a-protocol.org/latest/specification/, 2026. Accessed November 2026.

[2] S. Amershi, D. Weld, M. Vorvoreanu, A. Fourney, B. Nushi, P. Collisson, J. Suh, S. Iqbal, P. N. Bennett, K. Inkpen, J. Teevan, R. Kikin-Gil, and E. Horvitz. Guidelines for human-AI interaction. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2019.

[3] D. J. Anderson. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, Sequim, WA, 2010.

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

[5] D. Ariely and K. Wertenbroch. Procrastination, deadlines, and performance: Self-control by precommitment. *Psychological Science*, 13(3):219–224, 2002.

[6] Asana. Anatomy of Work Global Index 2023. Survey report, https://asana.com/resources/anatomy-of-work, 2023.

[7] D. H. Autor. Why are there still so many jobs? The history and future of workplace automation. *Journal of Economic Perspectives*, 29(3):3–30, 2015.

[8] S. Azhar. Building information modeling (BIM): Trends, benefits, risks, and challenges for the AEC industry. *Leadership and Management in Engineering*, 11(3):241–252, 2011.

[9] H. G. Ballard. *The Last Planner System of Production Control*. PhD thesis, University of Birmingham, 2000.

[10] J. Becker, N. Rush, E. Barnes, and D. Rein. Measuring the impact of early-2025 AI on experienced open-source developer productivity. arXiv:2507.09089, 2025.

[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, 1995. doi: 10.1145/223784.223785.

[12] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[13] M. Bloch, S. Blumberg, and J. Laartz. Delivering large-scale IT projects on time, on budget, and on value. McKinsey & Company, with the University of Oxford, 2012.

[14] A. Brand, L. Allen, M. Altman, M. Hlava, and J. Scott. Beyond authorship: Attribution, contribution, collaboration, and credit. *Learned Publishing*, 28(2):151–155, 2015.

[15] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[16] E. Brynjolfsson, D. Li, and L. Raymond. Generative AI at work. *The Quarterly Journal of Economics*, 140(2):889–942, 2025.

[17] M. Budiu, 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.

[18] R. Buehler, D. Griffin, and M. Ross. Exploring the "planning fallacy": Why people underestimate their task completion times. *Journal of Personality and Social Psychology*, 67(3):366–381, 1994.

[19] buildingSMART International. BIM Collaboration Format (BCF) 3.0. https://www.buildingsmart.org/bim-collaboration-format-bcf-3-0/, 2026. Accessed November 2026.

[20] C. Castelfranchi. Commitments: From individual intentions to groups and organizations. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*, pages 41–48, 1995.

[21] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW)*, pages 353–362, 2006.

[22] M. Cemri, M. Z. Pan, S. Yang, et al. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems*, 2025. arXiv:2503.13657.

[23] P.-C. Cheng, C. A. N. Soules, G. R. Goodson, and J. D. Strunk. Sybilproof reputation mechanisms. In *Proceedings of the ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems*, pages 128–132, 2005.

[24] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, pages 85–98, 2012.

[25] R. H. Coase. The nature of the firm. *Economica*, 4(16):386–405, 1937.

[26] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2–3):213–261, 1990.

[27] P. R. Cohen and H. J. Levesque. Teamwork. *Noûs*, 25(4):487–512, 1991.

[28] J. N. Cummings and S. Kiesler. Coordination costs and project outcomes in multi-university collaborations. *Research Policy*, 36(10):1620–1634, 2007.

[29] F. Dell'Acqua, E. McFowland, III, E. R. Mollick, et al. Navigating the jagged technological frontier: Field experimental evidence of the effects of AI on knowledge worker productivity and quality. Working Paper 24-013, Harvard Business School, 2023.

[30] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 918–934, 2019.

[31] V. Dibia. Designing Multi-Agent Systems. Self-published book, 2025.

[32] J. L. G. Dietz. *Enterprise Ontology: Theory and Methodology*. Springer, 2006.

[33] 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.

[34] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[35] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 399–407, 1989.

[36] T. Eloundou, S. Manning, P. Mishkin, and D. Rock. GPTs are GPTs: Labor market impact potential of LLMs. *Science*, 384 (6702):1306–1308, 2024.

[37] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2):213–253, 1980.

[38] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19:624–633, 1976. doi: 10.1145/360363.360369.

[39] European Parliament and Council of the European Union. Regulation (EU) 2024/1689 of 13 June 2024 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act). Official Journal of the European Union, 2024. Annex III, point 4.

[40] European Parliament and Council of the European Union. Directive (EU) 2024/2831 of 23 October 2024 on improving working conditions in platform work. Official Journal of the European Union, 2024.

[41] R. Fabian. *Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules*. Richard Fabian, 2018.

[42] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005.

[43] A. Filippas, J. J. Horton, and J. M. Golden. Reputation inflation. *Marketing Science*, 41(4):733–745, 2022.

[44] B. Flyvbjerg. What you should know about megaprojects and why: An overview. *Project Management Journal*, 45(2):6–19, 2014.

[45] B. Flyvbjerg and D. Gardner. *How Big Things Get Done*. Currency, New York, 2023.

[46] B. Fong and D. I. Spivak. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. Cambridge University Press, 2019.

[47] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge Computer Laboratory, 2004.

[48] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987.

[49] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[50] GitHub. About code owners. `https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners`, 2026. Accessed November 2026.

[51] E. M. Goldratt. *Critical Chain*. North River Press, Great Barrington, MA, 1997.

[52] P. M. Gollwitzer and P. Sheeran. Implementation intentions and goal achievement: A meta-analysis of effects and processes. In *Advances in Experimental Social Psychology*, volume 38, pages 69–119. Academic Press, 2006.

[53] 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.

[54] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. Not what you've signed up for: compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security (AISec)*, pages 79–90, 2023.

[55] B. J. Grosz and S. Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357, 1996.

[56] J. Grudin. Groupware and social dynamics: Eight challenges for developers. *Communications of the ACM*, 37(1):92–105, 1994.

[57] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, 1985.

[58] J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494, 2003.

[59] S. Hong et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[60] E. Horvitz. Principles of mixed-initiative user interfaces. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 159–166, 1999.

[61] International Labour Organization. *World Employment and Social Outlook 2021: The Role of Digital Labour Platforms in Transforming the World of Work*. International Labour Office, Geneva, 2021.

[62] ISO. ISO 16739-1:2018 Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries, part 1: Data schema. International Organization for Standardization, 2018.

[63] ISO. ISO 19650-1:2018 organization and digitization of information about buildings and civil engineering works, including building information modelling (BIM), information management using building information modelling, part 1: Concepts and principles. International Organization for Standardization, 2018.

[64] N. R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review*, 8(3):223–250, 1993.

[65] J. E. Kelley, Jr. and M. R. Walker. Critical-path planning and scheduling. In *Papers Presented at the Eastern Joint IRE-AIEE-ACM Computer Conference*, pages 160–173, 1959.

[66] K. C. Kellogg, M. A. Valentine, and A. Christin. Algorithms at work: The new contested terrain of control. *Academy of Management Annals*, 14(1):366–410, 2020.

[67] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. E. Moreira, J. D. Owens, M. Zalewski, and T. Mattson. Mathematical foundations of the GraphBLAS. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2016.

[68] A. Kittur, J. V. Nickerson, M. Bernstein, E. Gerber, A. Shaw, J. Zimmerman, M. Lease, and J. Horton. The future of crowd work. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, pages 1301–1318. ACM, 2013.

[69] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2): 213–226, 1981.

[70] S. Lack and P. Sobociński. Adhesive and quasiadhesive categories. *RAIRO – Theoretical Informatics and Applications*, 39(3): 511–545, 2005.

[71] LangChain. LangGraph overview. https://docs.langchain.com/oss/python/langgraph/overview, 2026. Accessed October 2026.

[72] M. K. Lee, D. Kusbit, E. Metsky, and L. Dabbish. Working with machines: The impact of algorithmic and data-driven management on human workers. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1603–1612. ACM, 2015.

[73] H. Macomber and G. A. Howell. Linguistic action: Contributing to the theory of lean construction. In *Proceedings of the 11th Annual Conference of the International Group for Lean Construction*, Virginia, USA, 2003.

[74] T. W. Malone. *The Future of Work: How the New Order of Business Will Shape Your Organization, Your Management Style, and Your Life*. Harvard Business School Press, Boston, 2004.

[75] T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

[76] T. W. Malone, J. Yates, and R. I. Benjamin. Electronic markets and electronic hierarchies. *Communications of the ACM*, 30(6): 484–497, 1987.

[77] Manna. Manna. https://manna.ac, 2026.

[78] R. Medina-Mora, T. Winograd, R. Flores, and F. Flores. The action workflow approach to workflow management technology. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW)*, pages 281–288, 1992.

[79] Microsoft. Will AI fix work? 2023 Work Trend Index annual report. Microsoft WorkLab, 2023.

[80] Microsoft. 2025: The year the Frontier Firm is born. 2025 Work Trend Index annual report. Microsoft WorkLab, 2025.

[81] L. Moreau and P. Missier. PROV-DM: The PROV Data Model. W3C Recommendation, 2013. 30 April 2013.

[82] S. Noy and W. Zhang. Experimental evidence on the productivity effects of generative artificial intelligence. *Science*, 381 (6654):187–192, 2023.

[83] Object Management Group. Business process model and notation (BPMN), version 2.0. OMG document formal/2011-01-03, 2011.

[84] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11:405–430, 1986. doi: 10.1145/7239.7265.

[85] E. Patterson, O. Lynch, and J. Fairbanks. Categorical data structures for technical computing. *Compositionality*, 4(5), 2022.

[86] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer. The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv:2302.06590, 2023.

[87] Project Management Institute. Pulse of the Profession 2020. Ahead of the Curve: Forging a future-focused culture. Project Management Institute, 2020.

[88] C. Qian et al. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

[89] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319, 1995.

[90] Reaktor. Agent guide. https://reaktor.build/docs/agent-guide, 2026.

[91] Reaktor. Agents: the agent plane. https://reaktor.build/docs/agent-plane, 2026.

[92] Reaktor. Reaktor framework overview. https://reaktor.build/docs/reaktor-framework-overview, 2026.

[93] Reaktor. The work graph in Reaktor: evidence, design history and build order. https://reaktor.build/docs/reaktor-work-graph, 2026.

[94] D. G. Reinertsen. *The Principles of Product Development Flow: Second Generation Lean Product Development*. Celeritas Publishing, Redondo Beach, CA, 2009.

[95] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000.

[96] D. Retelny, S. Robaszkiewicz, A. To, W. S. Lasecki, J. Patel, N. Rahmati, T. Doshi, M. Valentine, and M. S. Bernstein. Expert crowdsourcing with flash teams. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, pages 75–85. ACM, 2014.

[97] A. Rosenblat and L. Stark. Algorithmic labor and information asymmetries: A case study of Uber's drivers. *International Journal of Communication*, 10:3758–3784, 2016.

[98] K. Schwaber and J. Sutherland. The Scrum Guide. https://scrumguides.org, 2020.

[99] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011.

[100] Shibasis. Reaktor graph: Typed port hypergraphs, traversals and patches for multi-runtime systems. Design study, https://reaktor.build/docs/reaktor-v2, 2026. 8 October 2026. Not implemented.

[101] M. P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7(1):97–113, 1999.

[102] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, 1980.

[103] D. I. Spivak. Functorial data migration. *Information and Computation*, 217:31–51, 2012.

[104] M. Sporny et al. Verifiable Credentials Data Model v2.0. W3C Recommendation, https://www.w3.org/TR/vc-data-model-2.0/, 2025. 15 May 2025.

[105] M. Strathern. 'Improving ratings': audit in the British University system. *European Review*, 5(3):305–321, 1997.

[106] L. Suchman. Do categories have politics? The language/action perspective reconsidered. *Computer Supported Cooperative Work*, 2(3):177–190, 1994.

[107] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[108] The Bazel Authors. Sandboxing. https://bazel.build/docs/sandboxing, 2026. Accessed November 2026.

[109] The Kubernetes Authors. Controllers (Kubernetes documentation). https://kubernetes.io/docs/concepts/architecture/controller/, 2026. Accessed October 2026.

[110] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[111] M. Vaccaro, A. Almaatouq, and T. Malone. When combinations of humans and AI are useful: A systematic review and meta-analysis. *Nature Human Behaviour*, 8:2293–2303, 2024. doi:10.1038/s41562-024-02024-1.

[112] M. A. Valentine, D. Retelny, A. To, N. Rahmati, T. Doshi, and M. S. Bernstein. Flash organizations: Crowdsourcing complex work by structuring crowds as organizations. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 3523–3537. ACM, 2017.

[113] W. M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[114] 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.

[115] T. van Viegen et al. Neuromatch Academy: Teaching computational neuroscience with global accessibility. *Trends in Cognitive Sciences*, 25(7):535–538, 2021.

[116] G. Varghese and A. Lauck. Hashed and hierarchical timing wheels: Efficient data structures for implementing a timer facility. *IEEE/ACM Transactions on Networking*, 5(6):824–834, 1997.

[117] T. Winograd. A language/action perspective on the design of cooperative work. *Human–Computer Interaction*, 3(1):3–30, 1987.

[118] T. Winograd and F. Flores. *Understanding Computers and Cognition: A New Foundation for Design*. Ablex, 1986.

[119] A. J. Wood, M. Graham, V. Lehdonvirta, and I. Hjorth. Good gig, bad gig: Autonomy and algorithmic control in the global gig economy. *Work, Employment and Society*, 33(1):56–75, 2019.

[120] World Economic Forum. The Future of Jobs Report 2025. World Economic Forum, Geneva, 2025.

[121] Q. Wu et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv:2308.08155, 2023.

[122] S. Wuchty, B. F. Jones, and B. Uzzi. The increasing dominance of teams in production of knowledge. *Science*, 316(5827): 1036–1039, 2007.

[123] F. F. Xu, Y. Song, B. Li, et al. TheAgentCompany: Benchmarking LLM agents on consequential real world tasks. arXiv:2412.14161, 2024. Figures from the revised version; NeurIPS 2025.

[124] S. Yao, N. Shinn, P. Razavi, and K. Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, 2025. arXiv:2406.12045.

[125] P. Yolum and M. P. Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 527–534, 2002.

A Proofs

The proofs are short because each rests on a standard result. We state which one, check that its hypotheses hold, and write out the part that is specific to this paper.

A.1 The ambient category

Proof of Proposition 4. $\mathsf{URG} = \mathsf{Set}^{\mathcal{S}^{+}_{\mathsf{RG}}}$ is a category of functors from a small category to sets, and every such category is a topos. The slice of it over an object T is equivalent to the category of functors to Set from the category of elements of T, so it is a topos of the same form. Every topos is adhesive [70]. Attributes do not change this: with the types of attribute values fixed, graphs with attributes are again a slice of a functor category [85], and rules that compute new attribute values are covered by the standard theory of attributed graph transformation [34]. □

A.2 Independence

Throughout, read sets are sufficient and stable (Definition 1), and queries are evaluated deterministically, so that whether an evaluation stays within the synchronous budget depends only on the values it reads. Checks, acceptance queries and antecedents read structural predicates only, and regions contain structural predicates only. Three facts about patches are used.

(F1) A patch changes no predicate outside its write set: s and $P(s)$ agree on $\Pi \setminus Wr(P, s)$.

(F2) The structural predicates that a patch sets, and the values it gives them, are fixed by the patch. Each of them either belongs to an entity that the patch creates, or is the existence or the extent of an entity that the patch reads, or is the membership of an entity whose existence or extent the patch reads and sets.

(F3) Whether a patch applies is decided by predicates of $Rd(P)$.

(F1) holds because a patch changes only what it sets, and the write set contains every computed predicate whose value changes. (F2) holds because a patch is a list of assignments, names what it removes or replaces and so reads it, creates entities under identifiers of its own, and attaches or detaches an entity only by creating, removing or moving it, which sets its existence or its extent. (F3) is item (i) of the definition of $Rd(P)$ in Section 7. By (F2), two patches with the same base of which neither sets a predicate that the other reads set no structural predicate in common. They cannot create the same entity; if both set the existence or the extent of an entity present at the base, each reads what the other sets; and a membership is set only together with the existence or the extent of its member.

Lemma 5 (Frame for admission)**.** *Let P_1 be admissible at v with base v, and put $v_1 = P_1(v)$. Let P_2 be a patch with base v such that $Wr(P_1, v) \cap Rd(P_2) = \emptyset$ and $Wr(P_2, v) \cap Rd(P_1) = \emptyset$. Then P_2 is admissible at v if and only if it is admissible at v_1, and $P_2(P_1(v)) = P_1(P_2(v))$.*

Proof. Here $Rd(P_2)$ is the read set recorded at whichever of the two revisions P_2 is evaluated; the argument shows that the two coincide.

The state. By (F1), v and v_1 agree on $Rd(P_2)$. By (F3), P_2 applies at v_1 exactly when it applies at v, and by (F2) it sets the same structural predicates to the same values at both. In the same way, v and $P_2(v)$ agree on $Rd(P_1)$, so P_1 applies at $P_2(v)$ and sets there what it sets at v. The two patches set no structural predicate in common. In both $P_2(P_1(v))$ and $P_1(P_2(v))$, therefore, every structural predicate that P_1 sets has the value P_1 gives it, every one that P_2 sets has the value P_2 gives it, and every other has its value at v. A state is determined by its structural predicates, so $P_2(P_1(v)) = P_1(P_2(v))$. Computed predicates, thresholds and index predicates among them, are functions of the state and need no separate argument.

An observation used for several clauses. Let q be a query whose read set on $P_2(v)$ is part of $Rd(P_2)$. Then $P_2(v_1)$ and $P_2(v)$ agree on $Rd(q, P_2(v))$. That set contains structural predicates only. One that P_2 sets has the value P_2 gives it in both states. One that P_2 does not set has its value from v_1 in the first state and from v in the second, and these agree because the predicate lies in $Rd(P_2)$, which P_1 does not set. By sufficiency and stability, q has the same answer and the same read set on $P_2(v_1)$ as on $P_2(v)$, and by the assumption on evaluation it stays within the budget on one exactly when it does on the other. Call this $(*)$. The same argument with v_1 and v in place of $P_2(v_1)$ and $P_2(v)$ shows that a query whose read set at v lies in $Rd(P_2)$ has the same answer and read set at v_1 as at v.

(A1). Applicability was shown. Well-formedness and the subject space's checks are queries on the result whose read sets lie in $Rd(P_2)$, so by $(*)$ they give the same verdicts on $P_2(v_1)$ as on $P_2(v)$.

(A2). With head v_1 and base v, the only patch admitted in between is P_1, which sets no predicate of $Rd(P_2)$. With head v the clause is empty.

(A3). The predicates that say whether the grant of P_2 is in force and within budget, and the threshold predicates of the reservations in P_2, lie in $Rd(P_2)$ and have the same values at v_1 as at v. The grant's credential conditions are queries with read sets in $Rd(P_2)$, so they have the same answers. The grant covers what P_2 sets in both cases, because that is the same. In particular, if both patches reserve from one resource, P_1 has not made false the threshold predicate that P_2 read, so the reservation of P_2 still fits.

(A4). For each structural predicate π that P_2 sets, the lease index family of π and the extents of the commitments it names lie in $Rd(P_2)$, and P_1 sets none of them. The same commitments with the same terms are therefore consulted at v_1 as at v, and every other predicate the clause consults also lies in $Rd(P_2)$, so the verdict is the same.

(A5). For each structural predicate π that P_2 sets, the guard index family of π lies in $Rd(P_2)$, and P_1 sets none of its members. So the *conditional* and *active* commitments whose regions P_2 writes are the same at v_1 as at v, and their extents, which hold their states, terms and timeout policies, are the same. For each such commitment c, admission evaluates α and ψ before and after the change, and those read sets lie in $Rd(P_2)$. Before the change, the answers and the budget outcomes at v_1 equal those at v by the last sentence of $(*)$; after it, those on $P_2(v_1)$ equal those on $P_2(v)$ by $(*)$. Whether P_2 records an observation, amends or releases c is fixed by P_2. The verdict of the clause, and any *at risk* mark it sets, is therefore the same at both revisions. For every other commitment the clause asks nothing at either revision.

The read set of P_2 consists of the reads made while constructing it, which are the same at both revisions by the last sentence of $(*)$ and (F3), of the reads of checks and commitment queries, which are the same by $(*)$, and of the predicates consulted by (A3) and (A4) and the index families of what it sets, which were shown to be the same. So the two read sets coincide. □

Proof of Theorem 1. Apply Lemma 5 to the pair (P_1, P_2) and to the pair (P_2, P_1). □

Proof of Proposition 2. Every patch admitted between the base v and the head v' changes only predicates of its write set, by (F1), and by clause (A2) none of those lies in $Rd(P)$. So v' agrees with v on $Rd(P)$. The queries evaluated while P was constructed have read sets in $Rd(P)$; by sufficiency and stability they read the same values at v' and return the same answers, so constructing the change at v' gives the same patch. For the checks and the commitment queries, the argument for $(*)$ in the proof of Lemma 5 applies with v' in place of v_1. The lease and guard index families of what P sets lie in $Rd(P)$, so admission consults the same commitments at v' as at v; a commitment formed after v whose region meets what P sets would have changed a member of one of those families, and clause (A2) would have refused P. This gives (a). For (b), let $P_1, \dots, P_n$ be the admitted patches in order, and let h_i be the head when P_i was admitted. By what was just shown, P_i is the patch that the same change would have produced if constructed at h_i, and it passes there. The actual history is therefore the history in which each change is constructed and applied at the head, one after another. This is the validation argument of optimistic concurrency control [69], with predicates in place of records. □

Proof of Corollary 3. By Proposition 2 we may take the base of every patch to be the head at its admission. Let an interleaving of the patches $P_1, \dots, P_m$ of one task and $Q_1, \dots, Q_n$ of the other be given, and consider a patch Q of the second task that immediately follows a patch P of the first, with u the state before P. Since P sets no predicate that Q reads, Q constructed at u is the same patch as Q constructed at $P(u)$, as in the proof of Proposition 2. By Lemma 5 applied to (P, Q), Q is admissible at u exactly when it is admissible at $P(u)$, and the two orders give the same state; by the lemma applied to (Q, P), P is admissible at $Q(u)$ exactly when it is admissible at u. Exchanging the two therefore changes neither any verdict nor the state reached. Finitely many such exchanges turn the interleaving into $P_1, \dots, P_m, Q_1, \dots, Q_n$, and as many turn it into the sequence with the second task first. In the first of these the patches of the first task are admitted as they would be alone, and in the second those of the second task are. □

Shared capacity at the boundary. Let P_1 and P_2 reserve k_1 and k_2 of a resource r over overlapping intervals inside I, with $a = \mathrm{avail}(r, I, v)$. Each reads its own threshold predicate $\pi_{r,k_i,I}$. Admitting P_1 lowers the free amount to $a - k_1$, so it makes $\pi_{r,k_2,I}$ false exactly when $a - k_1 < k_2$. The two patches are therefore independent in this respect exactly when $k_1 + k_2 \le a$, and then both orders admit both. When $k_1 + k_2 > a$ while each $k_i \le a$, the first to commit makes false the threshold predicate the second read, so the second is refused under (A2), or under (A3) if it is checked again at the head, and r is never over-committed. The rule that only reservations read threshold predicates keeps this argument complete: a guard that read $\pi_{r,3,I}$ directly would be consulted by neither of two reservations of 3 and 7 from 10 free units, since neither alone changes that predicate, yet together they make it false (Section 7).

B Existing tools and the work graph

Table 9 compares kinds of tools that projects use today with the work graph. Each kind keeps part of the state that the engine records, and the engine can import it as records, proposals and subjects.

Kind of tool	What it keeps	Where it fits in the work graph	What the work graph adds
Issue trackers and task boards	tasks with assignees, states and due dates	proposed or active commitments; board columns as states	a creditor and an acceptance query; regions; admission
Critical-path schedulers	activities, durations and precedences	commitments with antecedents, run as a playbook	promises made by those who do the work; Percent Plan Complete with reasons
Coordination of building models	models, clash reports, issues on elements, approvals	a typed subject space; clash checks as acceptance queries; issues as records	commitments over regions of the model; admission before anything is built
Shared drives and documents	files, versions and comments	a subject space with declared read sets; comments as records	promises with acceptance; computed dependencies
Chat and email	messages, threads, decisions made in passing	records; decisions recorded as Decides with what they cited	status without asking; a queue ranked by what it threatens
Freelance marketplaces	listings, proposals, contracts, escrow and ratings	open commitments, proposals, reciprocal commitments, resources held for payment	milestones accepted against evidence; a reputation computed from evidence and carried between markets
Agent frameworks and protocols	steps, conversations, roles, tasks between agents	actors under grants with budgets; tasks as commitments	acceptance by another actor; obligations others can read; admission
Spreadsheets for events	guest lists, budgets, vendors and bookings	a subject space; vendors' commitments; bookings and money as resources	deadlines, dependencies and admission against bookings and budgets

Table 9: Kinds of tools compared with the work graph: what each keeps, where that state fits, and what the engine adds.

C Glossary

Acceptance query (α) Creditor-agreed query testing whether consequent φ holds; may run an automated check or require a creditor's `decide`.

Achieve and maintain (κ) Temporal commitment modalities (Definition 2): *achieve* promises φ holds by t; *maintain* promises φ stays true continuously until t (or $t = \infty$).

Actor A person, team, organisation, AI agent or service account; acts solely under Grants.

Admission Five-clause gate (Definition 3) checking validity, freshness of the read set, authority and escrow, write leases on R_0, and non-regression of guards over $R_0 \cup R_{\text{obs}}$.

Antecedent (ψ) Proposition whose truth makes a *conditional* commitment *active*.

Capability Trust-weighted query over an actor's historical discharged and violated commitments and their evidence (Section 5.3).

Commitment An addressable Commits hyperedge binding debtor d to creditor r (Definition 2).

Credential Signed Record from an issuing authority attesting to an actor's qualification until an expiry.

Decision Immutable Decides hyperedge adopting a proposition under a grant, citing considered records and optionally superseding an earlier decision.

Declared lease region (R_0) Predicates directly mutated by a commitment's work; locked exclusively for the debtor while the commitment is *active*.

Dependents The family of membership predicates of an entity, one for each entity that could be attached to it; listing what is attached reads the whole family, and attaching one entity sets one member (Definition 1).

Guard A standing *maintain* commitment whose debtor and creditor are the same actor and whose declared region is empty (Definition 2).

Index predicates ($\lambda_{\pi,c}$, $\gamma_{\pi,c}$) Computed predicates saying whether predicate π lies in the lease, or in a region, of commitment c; admission reads both families for every predicate a patch sets (Section 7).

Merge law A commutative, convergent update rule (CRDT or OT) that permits concurrent writes inside a shared R_0; guards still apply to what it writes.

Observed guard region ($R_{\text{obs}}(c, s)$) $Rd(\alpha, s) \cup Rd(\psi, s)$; open to any authorised writer provided α and a holding ψ do not regress under (A5).

Patch The form in which a change reaches the store: structural assignments and reservations with a base revision, an author and a grant. It reads what it removes or replaces, and the dependents of what it removes (Section 7).

Percent Plan Complete Share of commitments due in a window discharged on time, paired with structured violation causes.

Persistent chunk An immutable segment of a column shared across snapshots by copy-on-write: 4,096 rows in the C++ store and 1,024 in the Kotlin engine (Section 13).

Read set (Rd) Predicates inspected by a query or a patch; *declared* when listed by people, *recorded* when captured by the store's reading interface, *sufficient* when they determine the result, *stable* when every state that agrees on them gives the same read set again.

Review window (Δ_{rev}) Agreed interval during which a timely delivery `report` pauses the debtor's violation clock while awaiting the creditor's `decide`.

Shadow period Online migration phase evaluating changes on both old and new subject spaces before cutover (Section 10.5).

Threshold predicate ($\pi_{r,k,I}$) The predicate $\text{avail}(r, I, s) \geq k$; read only when a reservation is tested, so reservations from one resource do not conflict while enough remains.

Witness Structured refusal payload (c, π, d, r) naming the failed clause, predicate, commitment and holder.

Write set ($Wr(P, s)$) The predicates a patch sets: existence and extent of what it creates, removes or replaces, the memberships it changes, and the computed predicates whose values it changes.

D C++23 concurrent graph

This appendix reproduces the concurrent hypergraph store of Section 13.3 in full: five headers and the test program that checks them, read from the files that compile. The store needs a C++23 compiler and standard library and nothing else. `make check` in `code/cpp` builds the test program three times with `-Wall -Wextra -Wpedantic -Werror`: optimised, with the address and undefined-behaviour sanitizers, and with the thread sanitizer, and it runs each build. All 40 checks pass in all three (builds of 12 November 2026, run again on 16 November 2026). Figure 6 shows how the parts fit together, and Listing 1 lists the files with the listing that prints each.

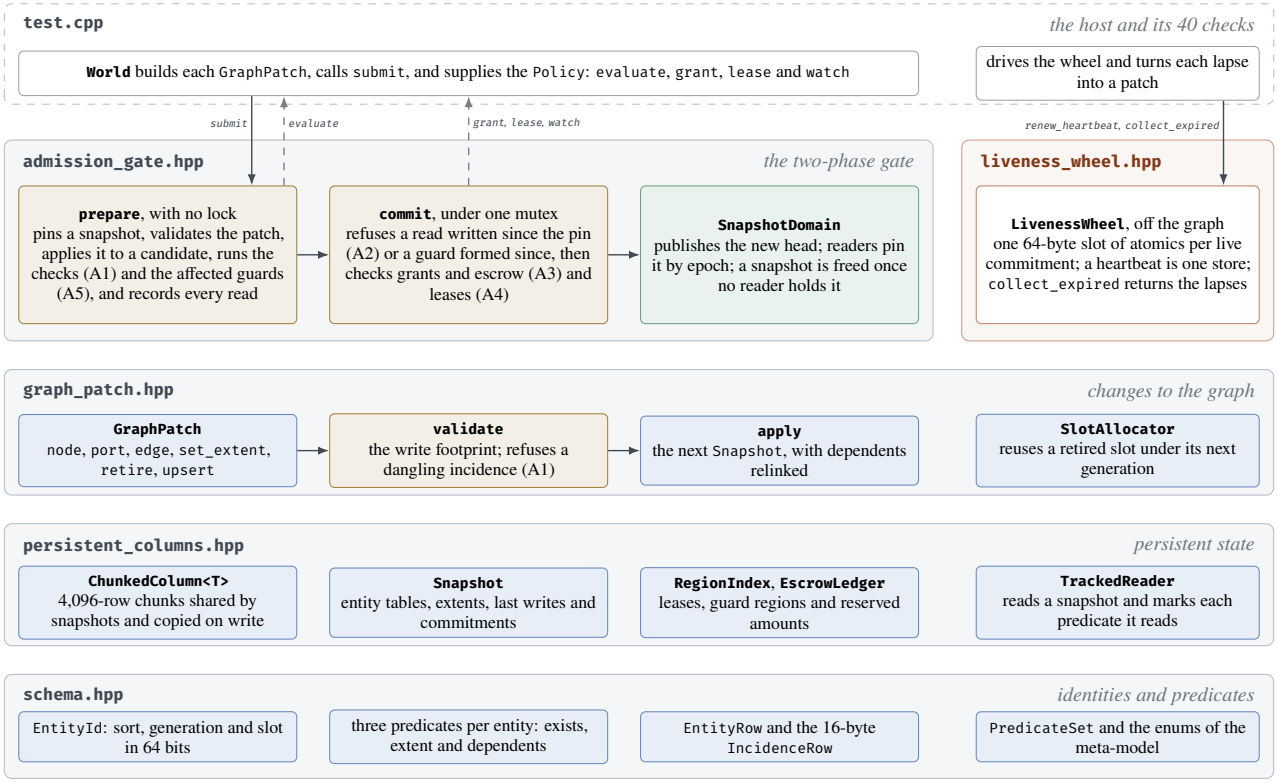

Figure 6: High-level design of the C++23 store. Each panel is one file. Each header includes the header in the band below it, and `liveness_wheel.hpp` includes only `schema.hpp`. A change enters at the top: the host builds a patch and submits it, `prepare` checks it on a pinned snapshot without a lock, `commit` checks under one short lock what may have changed since the pin, and `SnapshotDomain` publishes the result. Dashed arrows are calls back into the host: the store asks the host's `Policy` what a proposition means and whom to wake, and the host collects the lapses that the wheel finds. Amber marks the steps that can refuse a change.

code/cpp | **Listing 1.** The files of the store, in reading order. Files

```
code/cpp/
├── Makefile                          make check: three builds
├── test.cpp                Listing 7 the host and the 40 checks
└── work_graph/
    ├── schema.hpp          Listing 2 handles and predicates
    ├── persistent_columns.hpp Listing 3 columns, snapshots, reader
    ├── graph_patch.hpp     Listing 4 patches and slot reuse
    ├── admission_gate.hpp  Listing 5 epochs and the two-phase gate
    └── liveness_wheel.hpp  Listing 6 heartbeats off the graph
```

Reading order. Listing 2 defines entity handles and the three predicate identifiers of every entity: its existence, its extent and its dependents, where the last stands for the whole family of membership predicates with one last-write stamp. Listing 3 stores each sort in persistent 4,096-row columns, together with the incidence lists, the dependents lists, the commitment rows, the region index and the escrow ledger, and reads them only through `TrackedReader`. Listing 4 reuses retired slots under their next generation and turns a graph patch into its write footprint and its next snapshot, refusing patches that would leave a dangling incidence. Listing 5 pins snapshots with epochs and admits patches in two phases. Listing 6 keeps heartbeats off the graph. Listing 7 checks all of it.

What the host supplies. The store decides nothing about meaning. A `Policy` supplies four functions: how to evaluate a proposition, whether a grant authorises a proposal, whether an actor may write inside another commitment's lease, and whom to wake when predicates change. The test program supplies small versions of each; the Kotlin engine of Appendix E shows the full meaning.

work_graph/schema.hpp | **Listing 2.** Entity handles, the three predicates of every entity, and read sets. C++23

```cpp
#pragma once
#include <algorithm>
#include <bit>
#include <cstdint>
#include <optional>
#include <utility>
#include <vector>

namespace reaktor::wg {

enum class EntityTag : uint8_t { Node = 0, Port = 1, Edge = 2 };

struct EntityId {
  static constexpr uint32_t kGenerationMask = 0x3FFF'FFFFu;

  uint64_t raw = 0;

  static constexpr EntityId make(EntityTag tag, uint32_t gen, uint32_t slot) noexcept {
    return {(uint64_t(tag) << 62) | (uint64_t(gen & kGenerationMask) << 32) | slot};
  }
  constexpr EntityTag tag()        const noexcept { return EntityTag(raw >> 62); }
  constexpr uint32_t  generation() const noexcept { return uint32_t((raw >> 32) & kGenerationMask); }
  constexpr uint32_t  slot()       const noexcept { return uint32_t(raw & 0xFFFF'FFFFu); }
  constexpr bool      present()    const noexcept { return generation() != 0; }
  constexpr EntityId  successor()  const noexcept { return make(tag(), generation() % kGenerationMask + 1, slot()); }
  constexpr bool operator==(const EntityId&) const noexcept = default;
};

using RevisionId  = uint64_t;
using PredicateId = uint32_t;

enum class Aspect : uint8_t { Exists = 0, Extent = 1, Dependents = 2 };

constexpr PredicateId predicate_of(EntityId id, Aspect aspect) noexcept {
  return (id.slot() << 4) | (uint32_t(id.tag()) << 2) | uint32_t(aspect);
}

enum class MetaKind : uint8_t { Actor, Proposition, Record, Resource, Scope, Operator, State, Type };

enum class MetaFamily : uint8_t {
  Commits, Attests, Decides, Grants, Contributes, Uses,
  Bind, Session, Channel, Transition,
  Export, Access, References, Derives, Refines
};

enum class Modality      : uint8_t { Intended, Realised, Observed, Claimed };
enum class CommitKind    : uint8_t { Achieve, Maintain };
enum class CommitState   : uint8_t { Proposed, Conditional, Active, Discharged,
                                     Violated, Released, Cancelled, Declined };
enum class TimeoutPolicy : uint8_t { FailClosed, OptimisticAtRisk };

constexpr bool is_live(CommitState s) noexcept { return s == CommitState::Conditional || s == CommitState::Active; }

struct EntityRow {
  EntityId parent{};
  uint32_t generation = 0;
  uint16_t level      = 0;
  uint8_t  meta       = 0;
  Modality modality   = Modality::Realised;
  bool     live       = false;
};

struct IncidenceRow {
  EntityId endpoint;
  uint32_t edge_slot;
  uint16_t ordinal;
  uint8_t  role;
};
static_assert(sizeof(IncidenceRow) == 16);

class PredicateSet {
  std::vector<uint64_t> words_;

public:
  PredicateSet() : PredicateSet(64) {}
  explicit PredicateSet(size_t num_words) : words_(num_words, 0) {}

  void mark(PredicateId p) {
```

```
    const size_t w = p >> 6;
    if (w >= words_.size()) words_.resize(std::max(w + 1, words_.size() * 2), 0);
    words_[w] |= uint64_t{1} << (p & 63);
  }

  [[nodiscard]] bool contains(PredicateId p) const noexcept {
    const size_t w = p >> 6;
    return w < words_.size() && ((words_[w] >> (p & 63)) & 1) != 0;
  }

  void merge_from(const PredicateSet& other) {
    if (other.words_.size() > words_.size()) words_.resize(other.words_.size(), 0);
    for (size_t i = 0; i < other.words_.size(); ++i) words_[i] |= other.words_[i];
  }

  template <typename Fn>
  [[nodiscard]] std::optional<PredicateId> find_if(Fn&& fn) const {
    for (size_t w = 0; w < words_.size(); ++w)
      for (uint64_t bits = words_[w]; bits != 0; bits &= bits - 1)
        if (const auto p = PredicateId(w * 64 + std::countr_zero(bits)); fn(p)) return p;
    return std::nullopt;
  }

  template <typename Fn>
  [[nodiscard]] bool any_of(Fn&& fn) const { return find_if(std::forward<Fn>(fn)).has_value(); }

  [[nodiscard]] std::vector<PredicateId> ids() const {
    std::vector<PredicateId> out;
    (void)find_if([&out](PredicateId p) { out.push_back(p); return false; });
    return out;
  }
};

}
```

work_graph/persistent_columns.hpp | **Listing 3.** Persistent columns, entity tables, the region index and the tracked reader. C++23

```
#pragma once
#include "work_graph/schema.hpp"
#include <algorithm>
#include <array>
#include <memory>
#include <span>

namespace reaktor::wg {

template <typename T>
class ChunkedColumn {
  static constexpr size_t kChunkShift = 12;
  static constexpr size_t kChunkSize  = size_t{1} << kChunkShift;
  static constexpr size_t kChunkMask  = kChunkSize - 1;
  using Chunk = std::array<T, kChunkSize>;

  std::vector<std::shared_ptr<const Chunk>> chunks_;
  size_t size_ = 0;
  T      empty_{};

public:
  ChunkedColumn() = default;
  explicit ChunkedColumn(T empty) : empty_(std::move(empty)) {}

  [[nodiscard]] size_t size() const noexcept { return size_; }

  [[nodiscard]] const T& get(uint32_t slot) const noexcept {
    const size_t ci = slot >> kChunkShift;
    if (slot >= size_ || !chunks_[ci]) return empty_;
    return (*chunks_[ci])[slot & kChunkMask];
  }

  [[nodiscard]] ChunkedColumn with_updates(std::span<const std::pair<uint32_t, T>> edits) const {
    ChunkedColumn next = *this;
    std::vector<std::pair<size_t, std::shared_ptr<Chunk>>> fresh;
    for (const auto& [slot, val] : edits) {
      const size_t ci = slot >> kChunkShift;
      auto it = std::find_if(fresh.begin(), fresh.end(), [ci](const auto& f) { return f.first == ci; });
      if (it == fresh.end()) {
        auto copy = std::make_shared<Chunk>();
        if (ci < chunks_.size() && chunks_[ci]) *copy = *chunks_[ci]; else copy->fill(empty_);
        fresh.emplace_back(ci, std::move(copy));
        it = std::prev(fresh.end());
      }
      (*it->second)[slot & kChunkMask] = val;
      next.size_ = std::max(next.size_, size_t{slot} + 1);
    }
```

```
    for (auto& [ci, chunk] : fresh) {
      if (ci >= next.chunks_.size()) next.chunks_.resize(ci + 1);
      next.chunks_[ci] = std::move(chunk);
    }
    return next;
  }
};

inline constexpr uint32_t kNoSlot = UINT32_MAX;

struct CommitmentRow {
  EntityId      id{}, debtor{}, creditor{}, authority_grant{}, consequent{}, antecedent{};
  CommitKind    kind           = CommitKind::Achieve;
  CommitState   state          = CommitState::Proposed;
  TimeoutPolicy timeout_policy = TimeoutPolicy::FailClosed;
  bool          at_risk        = false;
  bool          in_review      = false;
  bool          delegated      = false;

  [[nodiscard]] constexpr bool holds_lease() const noexcept { return state == CommitState::Active && !delegated; }
};

struct Region {
  std::vector<PredicateId> lease{};
  std::vector<PredicateId> observed{};
};

using RegionRef     = std::shared_ptr<const Region>;
using WatcherList   = std::shared_ptr<const std::vector<uint32_t>>;
using IncidenceList = std::shared_ptr<const std::vector<IncidenceRow>>;
using EntityList    = std::shared_ptr<const std::vector<EntityId>>;

struct RegionIndex {
  ChunkedColumn<uint32_t>    lease_owner{kNoSlot};
  ChunkedColumn<WatcherList> guard_watchers;
  ChunkedColumn<RegionRef>   region_of;

  [[nodiscard]] std::vector<uint32_t> affected_guards(std::span<const PredicateId> writes) const {
    std::vector<uint32_t> hit;
    for (PredicateId p : writes) {
      if (const WatcherList& list = guard_watchers.get(p)) hit.insert(hit.end(), list->begin(), list->end());
    }
    std::sort(hit.begin(), hit.end());
    hit.erase(std::unique(hit.begin(), hit.end()), hit.end());
    return hit;
  }
};

struct EscrowLedger {
  ChunkedColumn<int64_t> reserved;
};

struct GraphTables {
  std::array<ChunkedColumn<EntityRow>, 3>  rows;
  ChunkedColumn<IncidenceList>             ends;
  std::array<ChunkedColumn<EntityList>, 3> dependents;
};

struct Snapshot {
  RevisionId                  revision = 0;
  GraphTables                 graph;
  ChunkedColumn<uint64_t>     extents;
  ChunkedColumn<RevisionId>   last_write;
  ChunkedColumn<CommitmentRow> commitments;
  EscrowLedger                escrow;
  RegionIndex                 regions;

  [[nodiscard]] const EntityRow* live(EntityId id) const noexcept {
    const EntityRow& row = graph.rows[size_t(id.tag())].get(id.slot());
    return row.live && row.generation == id.generation() ? &row : nullptr;
  }
};

class TrackedReader {
  const Snapshot& snap_;
  PredicateSet&   reads_;

public:
  TrackedReader(const Snapshot& snap, PredicateSet& reads) noexcept : snap_(snap), reads_(reads) {}

  [[nodiscard]] const Snapshot& snapshot() const noexcept { return snap_; }

  [[nodiscard]] const EntityRow* entity(EntityId id) {
    reads_.mark(predicate_of(id, Aspect::Exists));
    return snap_.live(id);
  }

```

```
  [[nodiscard]] uint64_t extent(EntityId id) {
    reads_.mark(predicate_of(id, Aspect::Extent));
    return entity(id) ? snap_.extents.get(predicate_of(id, Aspect::Extent)) : 0;
  }

  [[nodiscard]] std::span<const IncidenceRow> ends(EntityId edge) {
    if (edge.tag() != EntityTag::Edge || !entity(edge)) return {};
    const IncidenceList& list = snap_.graph.ends.get(edge.slot());
    return list ? std::span<const IncidenceRow>(*list) : std::span<const IncidenceRow>{};
  }

  [[nodiscard]] std::span<const EntityId> dependents(EntityId id) {
    reads_.mark(predicate_of(id, Aspect::Dependents));
    if (!snap_.live(id)) return {};
    const EntityList& list = snap_.graph.dependents[size_t(id.tag())].get(id.slot());
    return list ? std::span<const EntityId>(*list) : std::span<const EntityId>{};
  }

  [[nodiscard]] const CommitmentRow* commitment(EntityId edge) {
    reads_.mark(predicate_of(edge, Aspect::Extent));
    if (!entity(edge)) return nullptr;
    const CommitmentRow& row = snap_.commitments.get(edge.slot());
    return row.id == edge ? &row : nullptr;
  }
};

}
```

work_graph/graph_patch.hpp | **Listing 4.** Slot reuse under generations, and graph patches with the dangling condition. C++23

```
#pragma once
#include "work_graph/persistent_columns.hpp"
#include <atomic>
#include <expected>
#include <iterator>
#include <mutex>
#include <stdexcept>
#include <unordered_map>

namespace reaktor::wg {

enum class Clause : uint8_t { A1_Invalid, A2_StaleRead, A3_GrantOrEscrow, A4_Lease, A5_Guard };

struct RefusalWitness {
  Clause      failed_clause;
  PredicateId blocking_predicate = 0;
  EntityId    blocking_commitment{};
  EntityId    holder{};
};

class SlotAllocator {
  static constexpr uint32_t kSlots = uint32_t{1} << 28;

  std::array<std::atomic<uint32_t>, 3> next_{};
  std::array<std::atomic<size_t>, 3>   spare_{};
  std::mutex                           mu_;
  std::array<std::vector<EntityId>, 3> free_;

public:
  [[nodiscard]] EntityId allocate(EntityTag tag) {
    const size_t t = size_t(tag);
    if (spare_[t].load(std::memory_order_acquire) != 0) {
      std::scoped_lock lk(mu_);
      if (!free_[t].empty()) {
        const EntityId id = free_[t].back();
        free_[t].pop_back();
        spare_[t].store(free_[t].size(), std::memory_order_release);
        return id;
      }
    }
    const uint32_t slot = next_[t].fetch_add(1, std::memory_order_relaxed);
    if (slot >= kSlots) throw std::length_error("entity slots exhausted");
    return EntityId::make(tag, 1, slot);
  }

  void recycle(EntityId id) {
    std::scoped_lock lk(mu_);
    auto& list = free_[size_t(id.tag())];
    list.push_back(id.successor());
    spare_[size_t(id.tag())].store(list.size(), std::memory_order_release);
  }
};

struct Created {
```

```
  EntityId                 id;
  EntityRow                row;
  uint64_t                 extent;
  std::vector<IncidenceRow> ends;
};

struct CommitmentUpdate {
  CommitmentRow row;
  RegionRef     region;
};

class GraphPatch {
  SlotAllocator*                          slots_;
  std::vector<Created>                    created_;
  std::vector<std::pair<EntityId, uint64_t>> extents_;
  std::vector<EntityId>                   retired_;
  std::vector<CommitmentUpdate>           commitments_;

  using Links = std::unordered_map<uint64_t, std::pair<std::vector<EntityId>, std::vector<EntityId>>>;

  [[nodiscard]] const Created* find(EntityId id) const noexcept {
    for (const Created& c : created_) if (c.id == id) return &c;
    return nullptr;
  }

  [[nodiscard]] bool retires(EntityId id) const noexcept { return std::ranges::find(retired_, id) != retired_.end(); }

  EntityId create(EntityTag tag, uint8_t meta, EntityId parent, uint64_t extent) {
    const EntityId id = slots_->allocate(tag);
    created_.push_back({id, EntityRow{parent, id.generation(), 0, meta, Modality::Realised, true}, extent, {}});
    return id;
  }

  [[nodiscard]] static std::vector<EntityId> unique(std::vector<EntityId> ids) {
    std::ranges::sort(ids, {}, &EntityId::raw);
    ids.erase(std::ranges::unique(ids).begin(), ids.end());
    return ids;
  }

  [[nodiscard]] static std::vector<EntityId> references(const Created& c) {
    std::vector<EntityId> out;
    if (c.row.parent.present()) out.push_back(c.row.parent);
    for (const IncidenceRow& r : c.ends) out.push_back(r.endpoint);
    return unique(std::move(out));
  }

  [[nodiscard]] static std::vector<EntityId> references(const Snapshot& at, EntityId id) {
    std::vector<EntityId> out;
    const EntityRow& row = at.graph.rows[size_t(id.tag())].get(id.slot());
    if (row.parent.present()) out.push_back(row.parent);
    if (id.tag() == EntityTag::Edge)
      if (const IncidenceList& ends = at.graph.ends.get(id.slot())) for (const IncidenceRow& r : *ends)
          out.push_back(r.endpoint);
    return unique(std::move(out));
  }

  template <typename Row, typename Key>
  [[nodiscard]] static std::shared_ptr<const std::vector<Row>> relink(const std::shared_ptr<const std::vector<Row>>& old,
                                                                      const std::vector<Key>& added, const std::vector<Key>&
                                                                          removed) {
    std::vector<Row> list = old ? *old : std::vector<Row>{};
    std::erase_if(list, [&](const Row& r) { return std::ranges::find(removed, r) != removed.end(); });
    for (const Key& k : added) if (std::ranges::find(list, k) == list.end()) list.push_back(k);
    return list.empty() ? nullptr : std::make_shared<const std::vector<Row>>(std::move(list));
  }

  void apply_commitments(const Snapshot& at, Snapshot& next) const {
    std::vector<std::pair<uint32_t, CommitmentRow>> rows;
    std::vector<std::pair<uint32_t, RegionRef>> regions;
    std::vector<std::pair<uint32_t, uint32_t>> owners;
    std::unordered_map<PredicateId, std::pair<std::vector<uint32_t>, std::vector<uint32_t>>> watchers;
    for (const CommitmentUpdate& u : commitments_) {
      const uint32_t slot = u.row.id.slot();
      const CommitmentRow& old = at.commitments.get(slot);
      if (const RegionRef& region = at.regions.region_of.get(slot); region && old.id == u.row.id && is_live(old.state)) {
        if (old.holds_lease())
          for (PredicateId p : region->lease) if (at.regions.lease_owner.get(p) == slot) owners.emplace_back(p, kNoSlot);
        for (PredicateId p : region->lease) watchers[p].second.push_back(slot);
        for (PredicateId p : region->observed) watchers[p].second.push_back(slot);
      }
      if (is_live(u.row.state)) {
        if (u.row.holds_lease()) for (PredicateId p : u.region->lease) owners.emplace_back(p, slot);
        for (PredicateId p : u.region->lease) watchers[p].first.push_back(slot);
        for (PredicateId p : u.region->observed) watchers[p].first.push_back(slot);
      }
      rows.emplace_back(slot, u.row);
      regions.emplace_back(slot, u.region);
```

```
    }
    std::vector<std::pair<uint32_t, WatcherList>> lists;
    for (const auto& [p, change] : watchers) {
      auto& [added, removed] = change;
      std::vector<uint32_t> kept;
      std::ranges::copy_if(removed, std::back_inserter(kept), [&](uint32_t s) { return std::ranges::find(added, s) ==
          added.end(); });
      lists.emplace_back(p, relink(at.regions.guard_watchers.get(p), added, kept));
    }
    next.commitments            = at.commitments.with_updates(rows);
    next.regions.region_of      = at.regions.region_of.with_updates(regions);
    next.regions.lease_owner    = at.regions.lease_owner.with_updates(owners);
    next.regions.guard_watchers = at.regions.guard_watchers.with_updates(lists);
  }

public:
  explicit GraphPatch(SlotAllocator& slots) noexcept : slots_(&slots) {}

  EntityId node(MetaKind kind, EntityId parent = {}, uint64_t extent = 0) {
    return create(EntityTag::Node, uint8_t(kind), parent, extent);
  }

  EntityId port(EntityId owner, uint64_t extent = 0) {
    return create(EntityTag::Port, uint8_t(MetaKind::State), owner, extent);
  }

  EntityId edge(MetaFamily family, const std::vector<std::pair<uint8_t, EntityId>>& ends, EntityId scope = {}) {
    const EntityId id = create(EntityTag::Edge, uint8_t(family), scope, 0);
    auto& rows = created_.back().ends;
    for (const auto& [role, endpoint] : ends) {
      const auto ordinal = std::ranges::count_if(rows, [role](const IncidenceRow& r) { return r.role == role; });
      rows.push_back({endpoint, id.slot(), uint16_t(ordinal), role});
    }
    return id;
  }

  void set_extent(EntityId id, uint64_t value) { extents_.emplace_back(id, value); }
  void retire(EntityId id) { retired_.push_back(id); }
  void upsert(const CommitmentRow& row, Region region) {
    commitments_.push_back({row, std::make_shared<const Region>(std::move(region))});
  }

  [[nodiscard]] std::span<const CommitmentUpdate> commitments() const noexcept { return commitments_; }
  [[nodiscard]] std::span<const EntityId> retired() const noexcept { return retired_; }
  [[nodiscard]] bool upserts(EntityId c) const noexcept {
    return std::ranges::any_of(commitments_, [c](const CommitmentUpdate& u) { return u.row.id == c; });
  }

  void release() {
    for (const Created& c : created_) slots_->recycle(c.id);
    created_.clear();
  }

  [[nodiscard]] std::expected<std::vector<PredicateId>, RefusalWitness> validate(TrackedReader& at) const {
    std::vector<PredicateId> footprint;
    auto invalid = [](EntityId id, Aspect aspect = Aspect::Exists) {
      return std::unexpected(RefusalWitness{Clause::A1_Invalid, predicate_of(id, aspect)});
    };
    auto whole = [&footprint](EntityId id) {
      for (Aspect a : {Aspect::Exists, Aspect::Extent, Aspect::Dependents}) footprint.push_back(predicate_of(id, a));
    };
    for (const Created& c : created_) {
      if (c.id.tag() == EntityTag::Port && c.row.parent.tag() != EntityTag::Node) return invalid(c.id);
      whole(c.id);
      for (EntityId r : references(c)) {
        if (retires(r) || (!find(r) && !at.entity(r))) return invalid(r);
        footprint.push_back(predicate_of(r, Aspect::Dependents));
      }
    }
    for (const auto& [id, value] : extents_) {
      if (retires(id) || (!find(id) && !at.entity(id))) return invalid(id);
      footprint.push_back(predicate_of(id, Aspect::Extent));
    }
    for (EntityId id : retired_) {
      if (find(id) || !at.entity(id)) return invalid(id);
      for (EntityId d : at.dependents(id)) if (!retires(d)) return invalid(id, Aspect::Dependents);
      if (const CommitmentRow* c = id.tag() == EntityTag::Edge ? at.commitment(id) : nullptr; c && is_live(c->state)) return
          invalid(id, Aspect::Extent);
      whole(id);
      for (EntityId r : references(at.snapshot(), id)) footprint.push_back(predicate_of(r, Aspect::Dependents));
    }
    for (const CommitmentUpdate& u : commitments_) {
      const Created* made = find(u.row.id);
      const EntityRow* row = made ? &made->row : at.entity(u.row.id);
      if (!row || u.row.id.tag() != EntityTag::Edge || row->meta != uint8_t(MetaFamily::Commits)) return invalid(u.row.id);
      footprint.push_back(predicate_of(u.row.id, Aspect::Extent));
    }
```

```
    std::ranges::sort(footprint);
    footprint.erase(std::ranges::unique(footprint).begin(), footprint.end());
    return footprint;
  }

  [[nodiscard]] Snapshot apply(const Snapshot& at, std::span<const PredicateId> footprint, RevisionId revision) const {
    Snapshot next = at;
    next.revision = revision;
    std::array<std::vector<std::pair<uint32_t, EntityRow>>, 3> rows;
    std::vector<std::pair<uint32_t, IncidenceList>> ends;
    std::vector<std::pair<uint32_t, uint64_t>> extents;
    Links links;
    for (const Created& c : created_) {
      rows[size_t(c.id.tag())].emplace_back(c.id.slot(), c.row);
      extents.emplace_back(predicate_of(c.id, Aspect::Extent), c.extent);
      if (c.id.tag() == EntityTag::Edge) ends.emplace_back(c.id.slot(), std::make_shared<const
          std::vector<IncidenceRow>>(c.ends));
      for (EntityId r : references(c)) links[r.raw].first.push_back(c.id);
    }
    for (const auto& [id, value] : extents_) extents.emplace_back(predicate_of(id, Aspect::Extent), value);
    for (EntityId id : retired_) {
      for (EntityId r : references(at, id)) links[r.raw].second.push_back(id);
      EntityRow row = at.graph.rows[size_t(id.tag())].get(id.slot());
      row.live = false;
      rows[size_t(id.tag())].emplace_back(id.slot(), row);
      extents.emplace_back(predicate_of(id, Aspect::Extent), 0);
      if (id.tag() == EntityTag::Edge) ends.emplace_back(id.slot(), nullptr);
    }
    std::array<std::vector<std::pair<uint32_t, EntityList>>, 3> dependents;
    for (const auto& [raw, change] : links) {
      const EntityId holder{raw};
      const auto& column = at.graph.dependents[size_t(holder.tag())];
      dependents[size_t(holder.tag())].emplace_back(holder.slot(), relink(column.get(holder.slot()), change.first,
          change.second));
    }
    for (size_t t = 0; t < 3; ++t) {
      next.graph.rows[t]       = at.graph.rows[t].with_updates(rows[t]);
      next.graph.dependents[t] = at.graph.dependents[t].with_updates(dependents[t]);
    }
    next.graph.ends = at.graph.ends.with_updates(ends);
    next.extents    = at.extents.with_updates(extents);
    std::vector<std::pair<uint32_t, RevisionId>> stamps;
    for (PredicateId p : footprint) stamps.emplace_back(p, revision);
    next.last_write = at.last_write.with_updates(stamps);
    if (!commitments_.empty()) apply_commitments(at, next);
    return next;
  }
};

}
```

work_graph/admission_gate.hpp | **Listing 5.** Epoch-pinned snapshots and the two-phase admission gate. C++23

```
#pragma once
#include "work_graph/graph_patch.hpp"
#include <functional>

namespace reaktor::wg {

enum class EvalStatus : uint8_t { Holds, Fails, BudgetExceeded };

struct Reservation { EntityId resource; int64_t amount; };

struct Proposal {
  explicit Proposal(GraphPatch p, RevisionId base = 0, EntityId by = {}, EntityId under = {})
      : patch(std::move(p)), base_revision(base), actor(by), grant(under) {}

  GraphPatch               patch;
  RevisionId               base_revision;
  EntityId                 actor, grant;
  PredicateSet             author_reads;
  std::vector<Reservation> reservations;
  std::vector<EntityId>    required_checks;
  bool                     observation = false;
};

struct Policy {
  std::function<EvalStatus(TrackedReader&, EntityId)>                     evaluate;
  std::function<bool(const Snapshot&, const Proposal&)>                   grant;
  std::function<bool(const Snapshot&, const CommitmentRow&, const Proposal&)> lease;
  std::function<void(RevisionId, std::span<const PredicateId>)>           watch;
};

struct Guarded {
```

```
  std::vector<uint32_t>                          evaluated;
  std::vector<std::pair<uint32_t, CommitmentRow>> at_risk;
};

struct Prepared {
  explicit Prepared(Proposal p) : proposal(std::move(p)) {}

  Proposal                 proposal;
  RevisionId               pinned = 0;
  PredicateSet             reads;
  std::vector<PredicateId> footprint;
  Guarded                  guarded;
};

class SnapshotDomain {
  static constexpr uint64_t kQuiescent = UINT64_MAX;
  struct alignas(64) Announcement { std::atomic<uint64_t> epoch{kQuiescent}; };

  std::atomic<const Snapshot*>                    head_;
  std::atomic<uint64_t>                           epoch_{0};
  std::vector<Announcement>                       readers_;
  std::vector<std::pair<uint64_t, const Snapshot*>> retired_;

public:
  class Pin {
    Announcement*   slot_;
    const Snapshot* snap_;
  public:
    Pin(Announcement& slot, const Snapshot* snap) noexcept : slot_(&slot), snap_(snap) {}
    Pin(const Pin&) = delete;
    Pin& operator=(const Pin&) = delete;
    ~Pin() { slot_->epoch.store(kQuiescent, std::memory_order_release); }
    [[nodiscard]] const Snapshot& operator*()  const noexcept { return *snap_; }
    [[nodiscard]] const Snapshot* operator->() const noexcept { return snap_; }
  };

  SnapshotDomain(const Snapshot* initial, size_t max_readers) : head_(initial), readers_(max_readers) {}
  SnapshotDomain(const SnapshotDomain&) = delete;
  SnapshotDomain& operator=(const SnapshotDomain&) = delete;
  ~SnapshotDomain() {
    for (const auto& [epoch, snap] : retired_) delete snap;
    delete head_.load();
  }

  [[nodiscard]] Pin pin(size_t reader) noexcept {
    Announcement& slot = readers_[reader];
    slot.epoch.store(epoch_.load(std::memory_order_acquire), std::memory_order_seq_cst);
    return Pin(slot, head_.load(std::memory_order_seq_cst));
  }

  [[nodiscard]] const Snapshot& head() const noexcept { return *head_.load(std::memory_order_acquire); }

  void publish(const Snapshot* next) {
    const Snapshot* old = head_.exchange(next, std::memory_order_seq_cst);
    retired_.emplace_back(epoch_.fetch_add(1, std::memory_order_seq_cst), old);
    uint64_t oldest = epoch_.load(std::memory_order_seq_cst);
    for (const Announcement& r : readers_) oldest = std::min(oldest, r.epoch.load(std::memory_order_seq_cst));
    std::erase_if(retired_, [oldest](const auto& entry) {
      if (entry.first >= oldest) return false;
      delete entry.second;
      return true;
    });
  }
};

class WorkGraphStore {
  SnapshotDomain domain_;
  SlotAllocator  slots_;
  std::mutex     commit_mu_;
  Policy         policy_;

  [[nodiscard]] std::expected<Guarded, RefusalWitness> guard(const Snapshot& base, const Snapshot& candidate,
                                                             const Prepared& p, PredicateSet& reads) const {
    Guarded out{base.regions.affected_guards(p.footprint), {}};
    for (uint32_t slot : out.evaluated) {
      const CommitmentRow& c = base.commitments.get(slot);
      if (!is_live(c.state) || p.proposal.patch.upserts(c.id)) continue;
      reads.mark(predicate_of(c.id, Aspect::Extent));
      bool unknown = false;
      for (EntityId condition : {c.consequent, c.antecedent}) {
        if (!condition.present()) continue;
        PredicateSet before_reads, after_reads;
        TrackedReader before(base, before_reads), after(candidate, after_reads);
        const EvalStatus held = policy_.evaluate(before, condition);
        const EvalStatus holds = policy_.evaluate(after, condition);
        reads.merge_from(before_reads);
        reads.merge_from(after_reads);
```

```
        if (held == EvalStatus::BudgetExceeded || holds == EvalStatus::BudgetExceeded) {
          if (c.timeout_policy == TimeoutPolicy::FailClosed)
            return std::unexpected(RefusalWitness{Clause::A5_Guard, predicate_of(condition, Aspect::Extent), c.id,
                c.creditor});
          unknown = true;
        } else if (held == EvalStatus::Holds && holds == EvalStatus::Fails && !p.proposal.observation) {
          const auto hit = std::ranges::find_if(p.footprint, [&](PredicateId q) { return after_reads.contains(q); });
          const PredicateId blocking = hit != p.footprint.end() ? *hit : predicate_of(condition, Aspect::Extent);
          return std::unexpected(RefusalWitness{Clause::A5_Guard, blocking, c.id, c.debtor});
        }
      }
      if (unknown) {
        CommitmentRow marked = c;
        marked.at_risk = true;
        out.at_risk.emplace_back(slot, marked);
      }
    }
    return out;
  }

public:
  WorkGraphStore(const Snapshot* initial, size_t max_readers, Policy policy)
      : domain_(initial, max_readers), policy_(std::move(policy)) {}

  [[nodiscard]] SnapshotDomain::Pin pin(size_t reader) noexcept { return domain_.pin(reader); }
  [[nodiscard]] GraphPatch patch() noexcept { return GraphPatch(slots_); }

  [[nodiscard]] std::expected<Prepared, RefusalWitness> prepare(Proposal&& proposal, size_t reader) {
    Prepared p(std::move(proposal));
    auto refuse = [&p](RefusalWitness w) {
      p.proposal.patch.release();
      return std::unexpected(w);
    };
    auto base = domain_.pin(reader);
    p.pinned = base->revision;
    TrackedReader at(*base, p.reads);
    auto footprint = p.proposal.patch.validate(at);
    if (!footprint) return refuse(footprint.error());
    p.footprint = std::move(*footprint);
    for (const Reservation& r : p.proposal.reservations) {
      const EntityRow* row = at.entity(r.resource);
      if (!row || row->meta != uint8_t(MetaKind::Resource) || r.amount <= 0)
        return refuse({Clause::A1_Invalid, predicate_of(r.resource, Aspect::Exists)});
    }
    const Snapshot candidate = p.proposal.patch.apply(*base, p.footprint, base->revision + 1);
    TrackedReader after(candidate, p.reads);
    for (EntityId check : p.proposal.required_checks)
      if (policy_.evaluate(after, check) != EvalStatus::Holds) return refuse({Clause::A1_Invalid, predicate_of(check,
          Aspect::Extent)});
    auto guarded = guard(*base, candidate, p, p.reads);
    if (!guarded) return refuse(guarded.error());
    p.guarded = std::move(*guarded);
    for (const auto& [slot, row] : p.guarded.at_risk) p.footprint.push_back(predicate_of(row.id, Aspect::Extent));
    std::ranges::sort(p.footprint);
    p.footprint.erase(std::ranges::unique(p.footprint).begin(), p.footprint.end());
    return p;
  }

  [[nodiscard]] std::expected<RevisionId, RefusalWitness> commit(Prepared&& p) {
    auto refuse = [&p](RefusalWitness w) {
      p.proposal.patch.release();
      return std::unexpected(w);
    };
    const Proposal& prop = p.proposal;
    std::vector<std::pair<EntityId, int64_t>> demand;
    for (const Reservation& r : prop.reservations) {
      auto it = std::ranges::find(demand, r.resource, &std::pair<EntityId, int64_t>::first);
      if (it == demand.end()) demand.emplace_back(r.resource, r.amount); else it->second += r.amount;
    }

    std::scoped_lock lk(commit_mu_);
    const Snapshot& cur = domain_.head();
    if (auto q = prop.author_reads.find_if([&](PredicateId x) { return cur.last_write.get(x) > prop.base_revision; }))
      return refuse({Clause::A2_StaleRead, *q});
    if (auto q = p.reads.find_if([&](PredicateId x) { return cur.last_write.get(x) > p.pinned; }))
      return refuse({Clause::A2_StaleRead, *q});
    for (uint32_t slot : cur.regions.affected_guards(p.footprint)) {
      const CommitmentRow& c = cur.commitments.get(slot);
      if (is_live(c.state) && !std::ranges::binary_search(p.guarded.evaluated, slot) && !prop.patch.upserts(c.id))
        return refuse({Clause::A2_StaleRead, predicate_of(c.id, Aspect::Extent), c.id, c.debtor});
    }
    if (!policy_.grant(cur, prop))
      return refuse({Clause::A3_GrantOrEscrow, predicate_of(prop.grant, Aspect::Extent), {}, prop.actor});
    std::vector<std::pair<uint32_t, int64_t>> reserved;
    for (const auto& [resource, amount] : demand) {
      const int64_t capacity = int64_t(cur.extents.get(predicate_of(resource, Aspect::Extent)));
      const int64_t held = cur.escrow.reserved.get(resource.slot());
```

```
      if (capacity - held < amount) return refuse({Clause::A3_GrantOrEscrow, predicate_of(resource, Aspect::Extent)});
      reserved.emplace_back(resource.slot(), held + amount);
    }
    for (PredicateId q : p.footprint) {
      const uint32_t owner = cur.regions.lease_owner.get(q);
      if (owner == kNoSlot) continue;
      const CommitmentRow& holder = cur.commitments.get(owner);
      if (!policy_.lease(cur, holder, prop)) return refuse({Clause::A4_Lease, q, holder.id, holder.debtor});
    }
    for (const CommitmentUpdate& u : prop.patch.commitments()) {
      if (!u.row.holds_lease()) continue;
      for (PredicateId q : u.region->lease) {
        const uint32_t owner = cur.regions.lease_owner.get(q);
        if (owner == kNoSlot || owner == u.row.id.slot()) continue;
        const CommitmentRow& holder = cur.commitments.get(owner);
        return refuse({Clause::A4_Lease, q, holder.id, holder.debtor});
      }
    }

    auto* next = new Snapshot(prop.patch.apply(cur, p.footprint, cur.revision + 1));
    next->escrow.reserved = cur.escrow.reserved.with_updates(reserved);
    next->commitments     = next->commitments.with_updates(p.guarded.at_risk);
    const RevisionId published = next->revision;
    domain_.publish(next);
    for (EntityId id : prop.patch.retired()) slots_.recycle(id);
    if (policy_.watch) policy_.watch(published, p.footprint);
    return published;
  }

  [[nodiscard]] std::expected<RevisionId, RefusalWitness> submit(Proposal&& proposal, size_t reader) {
    auto prepared = prepare(std::move(proposal), reader);
    if (!prepared) return std::unexpected(prepared.error());
    return commit(std::move(*prepared));
  }
};

}
```

work_graph/liveness_wheel.hpp | **Listing 6.** The off-graph liveness wheel. C++23

```
#pragma once
#include "work_graph/schema.hpp"
#include <atomic>
#include <vector>

namespace reaktor::wg {

inline constexpr uint64_t kNever = UINT64_MAX;

struct alignas(64) LeaseSlot {
  std::atomic<uint64_t> commitment{0};
  std::atomic<uint64_t> deadline_ns{kNever};
  std::atomic<uint64_t> heartbeat_ivl_ns{kNever};
  std::atomic<uint64_t> review_window_ns{0};
  std::atomic<uint64_t> next_hb_due_ns{kNever};
  std::atomic<uint64_t> review_since_ns{kNever};
  std::atomic<bool>     active{false};
};
static_assert(sizeof(LeaseSlot) == 64);

enum class LivenessKind : uint8_t { HeartbeatLapsed, ReviewWindowExpired, DeadlineViolated };
struct LivenessEvent { EntityId commitment; LivenessKind kind; };

class LivenessWheel {
  std::vector<LeaseSlot> slots_;

  static constexpr uint64_t after(uint64_t now, uint64_t interval) noexcept {
    return interval >= kNever - now ? kNever : now + interval;
  }

public:
  explicit LivenessWheel(size_t max_active) : slots_(max_active) {}

  void activate(uint32_t i, EntityId c, uint64_t now_ns, uint64_t deadline_ns, uint64_t heartbeat_ns, uint64_t review_ns)
      noexcept {
    auto& s = slots_[i];
    s.commitment.store(c.raw, std::memory_order_relaxed);
    s.deadline_ns.store(deadline_ns, std::memory_order_relaxed);
    s.heartbeat_ivl_ns.store(heartbeat_ns, std::memory_order_relaxed);
    s.review_window_ns.store(review_ns, std::memory_order_relaxed);
    s.next_hb_due_ns.store(after(now_ns, heartbeat_ns), std::memory_order_relaxed);
    s.review_since_ns.store(kNever, std::memory_order_relaxed);
    s.active.store(true, std::memory_order_release);
  }
```

```

  void deactivate(uint32_t i) noexcept { slots_[i].active.store(false, std::memory_order_release); }

  void renew_heartbeat(uint32_t i, uint64_t now_ns) noexcept {
    auto& s = slots_[i];
    s.next_hb_due_ns.store(after(now_ns, s.heartbeat_ivl_ns.load(std::memory_order_relaxed)), std::memory_order_release);
  }

  void enter_review(uint32_t i, uint64_t now_ns) noexcept {
    slots_[i].review_since_ns.store(now_ns, std::memory_order_release);
  }

  void leave_review(uint32_t i, uint64_t now_ns) noexcept {
    auto& s = slots_[i];
    const uint64_t since = s.review_since_ns.exchange(kNever, std::memory_order_acq_rel);
    if (since == kNever) return;
    const uint64_t deadline = s.deadline_ns.load(std::memory_order_relaxed);
    if (deadline != kNever) s.deadline_ns.store(after(deadline, now_ns - since), std::memory_order_release);
    renew_heartbeat(i, now_ns);
  }

  [[nodiscard]] std::vector<LivenessEvent> collect_expired(uint64_t now_ns) const {
    std::vector<LivenessEvent> out;
    for (const auto& s : slots_) {
      if (!s.active.load(std::memory_order_acquire)) continue;
      const EntityId c{s.commitment.load(std::memory_order_relaxed)};
      const uint64_t since = s.review_since_ns.load(std::memory_order_acquire);
      if (since != kNever) {
        if (now_ns > after(since, s.review_window_ns.load(std::memory_order_relaxed))) out.push_back({c,
            LivenessKind::ReviewWindowExpired});
        continue;
      }
      if (now_ns > s.deadline_ns.load(std::memory_order_acquire)) {
        out.push_back({c, LivenessKind::DeadlineViolated});
      } else if (now_ns > s.next_hb_due_ns.load(std::memory_order_acquire)) {
        out.push_back({c, LivenessKind::HeartbeatLapsed});
      }
    }
    return out;
  }
};

}
```

test.cpp | **Listing 7.** The test program. C++23

```
#include "work_graph/schema.hpp"
#include "work_graph/persistent_columns.hpp"
#include "work_graph/graph_patch.hpp"
#include "work_graph/admission_gate.hpp"
#include "work_graph/liveness_wheel.hpp"
#include <cstdio>
#include <cstdlib>
#include <thread>

using namespace reaktor::wg;

static int failures = 0;

static void expect(bool ok, const char* what) {
  if (!ok) ++failures;
  std::printf("%s %s\n", ok ? "ok  " : "FAIL", what);
}

template <typename T>
static T must(std::expected<T, RefusalWitness> result) {
  if (!result) {
    std::printf("FAIL unexpected refusal, clause %d\n", int(result.error().failed_clause));
    std::exit(1);
  }
  return *result;
}

static bool refused(const std::expected<RevisionId, RefusalWitness>& r, Clause clause) {
  return !r && r.error().failed_clause == clause;
}

static bool contains(std::span<const EntityId> ids, EntityId id) { return std::ranges::find(ids, id) != ids.end(); }

using Query = std::function<EvalStatus(TrackedReader&)>;

struct World {
  struct Party { EntityId actor, grant; };

```

```
    std::unordered_map<uint64_t, Query> queries;
    std::atomic<int>                    watches{0};
    WorkGraphStore                      store;
    EntityId                            engine, scope;

    World() : store(new Snapshot(), 16, Policy{
        [this](TrackedReader& r, EntityId p) { return r.entity(p) ? queries.at(p.raw)(r) : EvalStatus::Fails; },
        [](const Snapshot& s, const Proposal& prop) {
          if (!prop.grant.present()) return true;
          const EntityRow* row = s.live(prop.grant);
          const IncidenceList& ends = s.graph.ends.get(prop.grant.slot());
          return row && row->meta == uint8_t(MetaFamily::Grants) && ends && ends->size() > 1 &&
                 (*ends)[1].endpoint == prop.actor && s.extents.get(predicate_of(prop.grant, Aspect::Extent)) == 0;
        },
        [](const Snapshot&, const CommitmentRow& holder, const Proposal& prop) { return holder.debtor == prop.actor; },
        [this](RevisionId, std::span<const PredicateId>) { watches.fetch_add(1, std::memory_order_relaxed); }}) {
      must(submit({}, [&](Proposal& p) {
        scope = p.patch.node(MetaKind::Scope);
        engine = p.patch.node(MetaKind::Actor, scope);
      }));
    }

    RevisionId head() { return store.pin(0)->revision; }

    template <typename Build>
    std::expected<RevisionId, RefusalWitness> submit(Party by, Build&& build, size_t reader = 0) {
      Proposal p(store.patch(), store.pin(reader)->revision, by.actor, by.grant);
      build(p);
      return store.submit(std::move(p), reader);
    }

    Party party() {
      Party made;
      must(submit({}, [&](Proposal& p) {
        made.actor = p.patch.node(MetaKind::Actor, scope);
        made.grant = p.patch.edge(MetaFamily::Grants, {{0, engine}, {1, made.actor}, {2, scope}}, scope);
      }));
      return made;
    }

    EntityId element(uint64_t extent) {
      EntityId made;
      must(submit({}, [&](Proposal& p) { made = p.patch.node(MetaKind::State, scope, extent); }));
      return made;
    }

    EntityId proposition(Query query) {
      EntityId made;
      must(submit({}, [&](Proposal& p) { made = p.patch.node(MetaKind::Proposition, scope); }));
      queries.emplace(made.raw, std::move(query));
      return made;
    }

    std::vector<PredicateId> observed(EntityId proposition) {
      PredicateSet reads;
      auto pin = store.pin(0);
      TrackedReader r(*pin, reads);
      (void)store_evaluate(r, proposition);
      return reads.ids();
    }

    EvalStatus store_evaluate(TrackedReader& r, EntityId p) { return r.entity(p) ? queries.at(p.raw)(r) : EvalStatus::Fails; }

    std::expected<RevisionId, RefusalWitness> promise(EntityId& c, Party debtor, Party creditor, EntityId consequent, CommitKind
        kind,
                                                      std::vector<PredicateId> lease, TimeoutPolicy timeout =
                                                          TimeoutPolicy::FailClosed) {
      const auto observing = observed(consequent);
      return submit(debtor, [&](Proposal& p) {
        c = p.patch.edge(MetaFamily::Commits, {{0, debtor.actor}, {1, creditor.actor}, {2, consequent}, {3, debtor.grant}},
            scope);
        p.patch.upsert(CommitmentRow{.id = c, .debtor = debtor.actor, .creditor = creditor.actor, .authority_grant =
            debtor.grant,
                                     .consequent = consequent, .kind = kind, .state = CommitState::Active, .timeout_policy =
                                         timeout},
                       Region{std::move(lease), observing});
      });
    }

    EntityId commitment(Party debtor, Party creditor, EntityId consequent, CommitKind kind, std::vector<PredicateId> lease = {},
                        TimeoutPolicy timeout = TimeoutPolicy::FailClosed) {
      EntityId c;
      must(promise(c, debtor, creditor, consequent, kind, std::move(lease), timeout));
      return c;
    }
  };
```

```
static Query differs_from(EntityId x, uint64_t bad) {
  return [x, bad](TrackedReader& r) { return r.extent(x) != bad ? EvalStatus::Holds : EvalStatus::Fails; };
}

static void graph_shape() {
  World w;
  auto gc = w.party(), sub = w.party();
  const EntityId done = w.proposition([](TrackedReader&) { return EvalStatus::Fails; });
  EntityId c, uses, decision, superseding;
  must(w.submit({}, [&](Proposal& p) {
    const EntityId budget = p.patch.node(MetaKind::Resource, w.scope, 1'000);
    c = p.patch.edge(MetaFamily::Commits, {{0, sub.actor}, {1, gc.actor}, {2, done}}, w.scope);
    uses = p.patch.edge(MetaFamily::Uses, {{0, c}, {1, budget}}, w.scope);
    decision = p.patch.edge(MetaFamily::Decides, {{0, gc.actor}, {1, done}, {2, gc.grant}}, w.scope);
    superseding = p.patch.edge(MetaFamily::Decides, {{0, gc.actor}, {1, done}, {2, gc.grant}, {3, decision}}, w.scope);
  }));
  auto pin = w.store.pin(0);
  PredicateSet reads;
  TrackedReader r(*pin, reads);
  expect(r.ends(uses)[0].endpoint == c && c.tag() == EntityTag::Edge, "a Uses edge ends on a Commits edge (higher-order
      incidence)");
  expect(contains(r.dependents(c), uses), "the Commits edge lists the Uses edge among its dependents");
  expect(contains(r.dependents(decision), superseding), "a Decides edge ends on the Decides edge it supersedes");
  expect(r.dependents(gc.grant).size() == 2, "both decisions cite the grant edge as their authority");
  expect(reads.contains(predicate_of(c, Aspect::Dependents)) && reads.contains(predicate_of(uses, Aspect::Exists)),
         "every read lands in the read set");
}

static void dangling_and_generations() {
  World w;
  auto a = w.party(), b = w.party();
  EntityId x, link;
  must(w.submit({}, [&](Proposal& p) {
    x = p.patch.node(MetaKind::State, w.scope, 7);
    link = p.patch.edge(MetaFamily::References, {{0, a.actor}, {1, x}}, w.scope);
  }));
  auto alone = w.submit({}, [&](Proposal& p) { p.patch.retire(x); });
  expect(refused(alone, Clause::A1_Invalid) && alone.error().blocking_predicate == predicate_of(x, Aspect::Dependents),
         "retiring a node that an edge still ends on is refused (dangling condition)");
  must(w.submit({}, [&](Proposal& p) {
    p.patch.retire(link);
    p.patch.retire(x);
  }));
  EntityId reused;
  must(w.submit({}, [&](Proposal& p) { reused = p.patch.node(MetaKind::State, w.scope, 9); }));
  expect(reused.slot() == x.slot() && reused.generation() == x.generation() + 1, "a retired slot is reused under the next
      generation");
  auto stale = w.submit({}, [&](Proposal& p) { p.patch.edge(MetaFamily::References, {{0, b.actor}, {1, x}}, w.scope); });
  expect(refused(stale, Clause::A1_Invalid) && stale.error().blocking_predicate == predicate_of(x, Aspect::Exists),
         "an edge to the stale handle is refused although its slot is live again");
}

static void freshness() {
  World w;
  auto a = w.party(), b = w.party();
  const EntityId x = w.element(1), y = w.element(2);
  const RevisionId base = w.head();
  Proposal reader(w.store.patch(), base, a.actor, a.grant);
  {
    auto pin = w.store.pin(0);
    TrackedReader r(*pin, reader.author_reads);
    reader.patch.set_extent(y, r.extent(x) + 1);
  }
  must(w.submit(b, [&](Proposal& p) { p.patch.set_extent(x, 5); }));
  auto stale = w.store.submit(std::move(reader), 0);
  expect(refused(stale, Clause::A2_StaleRead) && stale.error().blocking_predicate == predicate_of(x, Aspect::Extent),
         "a change whose author read a value written since its base is refused (A2)");
  Proposal unrelated(w.store.patch(), base, a.actor, a.grant);
  {
    auto pin = w.store.pin(0);
    TrackedReader r(*pin, unrelated.author_reads);
    unrelated.patch.set_extent(y, r.extent(y) + 1);
  }
  expect(w.store.submit(std::move(unrelated), 0).has_value(), "an unrelated change on the same old base is admitted");
}

static void phantoms_and_hubs() {
  World w;
  auto a = w.party(), b = w.party();
  const EntityId hub = w.element(0);
  const RevisionId base = w.head();
  auto link = [&](World::Party by) {
    Proposal p(w.store.patch(), base, by.actor, by.grant);
    p.patch.edge(MetaFamily::References, {{0, by.actor}, {1, hub}}, w.scope);
    return w.store.submit(std::move(p), 0);
  };
  const bool both = link(a).has_value() && link(b).has_value();
```

```
  expect(both, "two edges onto one hub from the same base are both admitted: appends commute");
  Proposal census(w.store.patch(), 0, a.actor, a.grant);
  {
    auto pin = w.store.pin(0);
    census.base_revision = pin->revision;
    TrackedReader r(*pin, census.author_reads);
    census.patch.set_extent(hub, r.dependents(hub).size());
  }
  must(link(b));
  auto phantom = w.store.submit(std::move(census), 0);
  expect(refused(phantom, Clause::A2_StaleRead) && phantom.error().blocking_predicate == predicate_of(hub, Aspect::Dependents),
         "a census of the hub's edges is refused once a new edge lands: no phantoms");
}

static void escrow() {
  World w;
  auto a = w.party();
  EntityId lift;
  must(w.submit({}, [&](Proposal& p) { lift = p.patch.node(MetaKind::Resource, w.scope, 2); }));
  const RevisionId base = w.head();
  auto reserve = [&] {
    Proposal p(w.store.patch(), base, a.actor, a.grant);
    p.reservations.push_back({lift, 1});
    return w.store.submit(std::move(p), 0);
  };
  auto r1 = reserve(), r2 = reserve(), r3 = reserve();
  expect(r1 && r2, "two reservations on the same base are both admitted: no hot-spot abort");
  expect(refused(r3, Clause::A3_GrantOrEscrow) && r3.error().blocking_predicate == predicate_of(lift, Aspect::Extent),
         "a third reservation is refused: the capacity of 2 is used up (A3)");
}

static void guards() {
  World w;
  auto owner = w.party(), mech = w.party();
  const EntityId x = w.element(1);
  const EntityId c = w.commitment(owner, owner, w.proposition(differs_from(x, 99)), CommitKind::Maintain);
  auto broken = w.submit(mech, [&](Proposal& p) { p.patch.set_extent(x, 99); });
  expect(refused(broken, Clause::A5_Guard) && broken.error().blocking_predicate == predicate_of(x, Aspect::Extent) &&
             broken.error().blocking_commitment == c && broken.error().holder == owner.actor,
         "a write that breaks a standing commitment is refused with a witness (A5)");
  expect(w.submit(mech, [&](Proposal& p) { p.patch.set_extent(x, 2); }).has_value(), "a write that keeps the invariant is admitted");
  auto seen = w.submit(mech, [&](Proposal& p) {
    p.patch.set_extent(x, 99);
    p.observation = true;
  });
  expect(seen.has_value(), "an observation is admitted even when it falsifies the invariant");

  const EntityId y = w.element(1), z = w.element(1);
  auto slow = [](EntityId e) { return [e](TrackedReader& r) { return r.extent(e) > 0 ? EvalStatus::BudgetExceeded : EvalStatus::Holds; }; };
  const EntityId closed = w.commitment(owner, owner, w.proposition(slow(y)), CommitKind::Achieve);
  auto timed_out = w.submit(mech, [&](Proposal& p) { p.patch.set_extent(y, 3); });
  expect(refused(timed_out, Clause::A5_Guard) && timed_out.error().blocking_commitment == closed && timed_out.error().holder == owner.actor,
         "a guard over its budget refuses the change when it fails closed");
  const EntityId open = w.commitment(owner, owner, w.proposition(slow(z)), CommitKind::Achieve, {}, TimeoutPolicy::OptimisticAtRisk);
  expect(w.submit(mech, [&](Proposal& p) { p.patch.set_extent(z, 3); }).has_value() && w.store.pin(0)->commitments.get(open.slot()).at_risk,
         "an optimistic guard over its budget admits the change and marks the commitment at risk");
}

static void leases() {
  World w;
  auto gc = w.party(), elec = w.party(), mech = w.party();
  const EntityId tray = w.element(10);
  const EntityId placed = w.proposition([tray](TrackedReader& r) { return r.extent(tray) == 11 ? EvalStatus::Holds : EvalStatus::Fails; });
  const EntityId c = w.commitment(elec, gc, placed, CommitKind::Achieve, {predicate_of(tray, Aspect::Extent)});
  auto intruder = w.submit(mech, [&](Proposal& p) { p.patch.set_extent(tray, 12); });
  expect(refused(intruder, Clause::A4_Lease) && intruder.error().blocking_commitment == c && intruder.error().holder == elec.actor,
         "a write into another debtor's lease is refused (A4)");
  expect(w.submit(elec, [&](Proposal& p) { p.patch.set_extent(tray, 11); }).has_value(), "the debtor writes inside its own lease");
  EntityId rival;
  auto second = w.promise(rival, mech, gc, placed, CommitKind::Achieve, {predicate_of(tray, Aspect::Extent)});
  expect(refused(second, Clause::A4_Lease) && second.error().blocking_commitment == c, "a second commitment cannot lease the same predicate");
  must(w.submit(elec, [&](Proposal& p) {
    CommitmentRow row = w.store.pin(0)->commitments.get(c.slot());
    row.state = CommitState::Discharged;
    p.patch.upsert(row, Region{});
  }));
  expect(w.submit(mech, [&](Proposal& p) { p.patch.set_extent(tray, 12); }).has_value(), "discharging the commitment frees its lease and its guard");
```

```
}

static void formed_guards() {
  World w;
  auto owner = w.party(), mech = w.party();
  const EntityId x = w.element(1);
  const EntityId sound = w.proposition(differs_from(x, 99));
  Proposal late(w.store.patch(), w.head(), mech.actor, mech.grant);
  late.patch.set_extent(x, 99);
  auto prepared = w.store.prepare(std::move(late), 0);
  expect(prepared.has_value(), "phase 1 accepts the write while nothing guards x");
  const EntityId c = w.commitment(owner, owner, sound, CommitKind::Maintain);
  auto committed = w.store.commit(std::move(*prepared));
  expect(refused(committed, Clause::A2_StaleRead) && committed.error().blocking_commitment == c,
         "phase 2 refuses it: a guard formed over x after phase 1 pinned its snapshot");
}

static void revoked_grants() {
  World w;
  auto a = w.party();
  const EntityId x = w.element(1);
  expect(w.submit(a, [&](Proposal& p) { p.patch.set_extent(x, 2); }).has_value(), "a live grant admits the change");
  must(w.submit({}, [&](Proposal& p) { p.patch.set_extent(a.grant, 1); }));
  auto revoked = w.submit(a, [&](Proposal& p) { p.patch.set_extent(x, 3); });
  expect(refused(revoked, Clause::A3_GrantOrEscrow) && revoked.error().holder == a.actor, "a change under a revoked grant is
      refused (A3)");
}

static void concurrency() {
  World w;
  constexpr size_t kWriters = 4, kEdges = 400, kIncrements = 200;
  std::vector<World::Party> writers;
  std::vector<EntityId> cells;
  for (size_t i = 0; i < kWriters; ++i) {
    writers.push_back(w.party());
    cells.push_back(w.element(0));
  }
  const EntityId hub = w.element(0), counter = w.element(0);

  std::atomic<bool> stop{false};
  std::atomic<long> pins{0}, torn{0}, refusals{0}, retries{0};
  std::vector<std::thread> readers;
  for (size_t t = 0; t < 2; ++t) {
    readers.emplace_back([&, t] {
      RevisionId last = 0;
      while (!stop.load(std::memory_order_relaxed)) {
        auto snap = w.store.pin(8 + t);
        PredicateSet reads;
        TrackedReader r(*snap, reads);
        size_t edges = 0;
        const auto& rows = snap->graph.rows[size_t(EntityTag::Edge)];
        for (uint32_t slot = 0; slot < rows.size(); ++slot) {
          const EntityRow& row = rows.get(slot);
          if (!row.live || row.meta != uint8_t(MetaFamily::References)) continue;
          for (const IncidenceRow& end : r.ends(EntityId::make(EntityTag::Edge, row.generation, slot)))
            if (!r.entity(end.endpoint)) torn.fetch_add(1);
          ++edges;
        }
        if (snap->revision < last || edges != r.dependents(hub).size()) torn.fetch_add(1);
        last = snap->revision;
        pins.fetch_add(1, std::memory_order_relaxed);
      }
    });
  }
  std::vector<std::thread> threads;
  for (size_t i = 0; i < kWriters; ++i) {
    threads.emplace_back([&, i] {
      for (size_t k = 1; k <= kEdges; ++k) {
        auto admitted = w.submit(writers[i], [&](Proposal& p) {
          const EntityId n = p.patch.node(MetaKind::State, w.scope, k);
          p.patch.edge(MetaFamily::References, {{0, n}, {1, hub}}, w.scope);
          p.patch.set_extent(cells[i], k);
        }, i);
        if (!admitted) refusals.fetch_add(1);
      }
      for (size_t k = 0; k < kIncrements; ++k) {
        for (;;) {
          Proposal p(w.store.patch(), 0, writers[i].actor, writers[i].grant);
          {
            auto pin = w.store.pin(i);
            p.base_revision = pin->revision;
            TrackedReader r(*pin, p.author_reads);
            p.patch.set_extent(counter, r.extent(counter) + 1);
          }
          if (w.store.submit(std::move(p), i)) break;
          retries.fetch_add(1);
        }
```

```
      }
    });
  }
  for (auto& t : threads) t.join();
  stop = true;
  for (auto& t : readers) t.join();

  auto head = w.store.pin(0);
  PredicateSet reads;
  TrackedReader r(*head, reads);
  expect(refusals == 0 && r.dependents(hub).size() == kWriters * kEdges, "four writers add 1,600 edges onto one hub with no refusal");
  expect(std::ranges::all_of(cells, [&](EntityId c) { return r.extent(c) == kEdges; }), "each writer's own cell holds its last write");
  expect(r.extent(counter) == kWriters * kIncrements, "a contended read-modify-write loses no increment");
  expect(torn == 0, "readers never saw a torn graph or a revision going backwards");
  std::printf("     reader pins %ld, counter retries %ld, watches %d\n", pins.load(), retries.load(), w.watches.load());
}

static std::vector<LivenessKind> kinds(const std::vector<LivenessEvent>& events) {
  std::vector<LivenessKind> out;
  for (const auto& e : events) out.push_back(e.kind);
  return out;
}

static void liveness() {
  LivenessWheel wheel(8);
  using enum LivenessKind;
  expect(wheel.collect_expired(1'000).empty(), "inactive slots never expire");
  wheel.activate(0, EntityId::make(EntityTag::Edge, 1, 42), 0, 1'000, 100, 50);
  expect(kinds(wheel.collect_expired(150)) == std::vector{HeartbeatLapsed}, "a quiet debtor lapses its heartbeat");
  wheel.renew_heartbeat(0, 150);
  expect(wheel.collect_expired(200).empty(), "a report renews the heartbeat with one atomic store");
  wheel.enter_review(0, 220);
  expect(wheel.collect_expired(260).empty(), "the debtor's clock pauses while the creditor reviews");
  expect(kinds(wheel.collect_expired(271)) == std::vector{ReviewWindowExpired}, "a review left past its window escalates");
  wheel.leave_review(0, 280);
  expect(kinds(wheel.collect_expired(1'050)) == std::vector{HeartbeatLapsed}, "a rejection resumes the clock with the deadline moved by the pause");
  expect(kinds(wheel.collect_expired(1'061)) == std::vector{DeadlineViolated}, "the moved deadline still binds");

  std::atomic<bool> stop{false};
  std::vector<std::thread> debtors;
  for (uint32_t i = 1; i <= 4; ++i) {
    wheel.activate(i, EntityId::make(EntityTag::Edge, 1, 100 + i), 0, kNever, 10, 0);
    debtors.emplace_back([&, i] {
      for (uint64_t now = 1; !stop.load(std::memory_order_relaxed); ++now) wheel.renew_heartbeat(i, now);
    });
  }
  size_t collected = 0;
  for (uint64_t now = 0; now < 2'000; ++now) collected += wheel.collect_expired(now).size();
  stop = true;
  for (auto& t : debtors) t.join();
  expect(wheel.collect_expired(kNever - 1).size() == 5, "renewals race the collector without locks, and every slot still lapses");
  for (uint32_t i = 0; i <= 4; ++i) wheel.deactivate(i);
  expect(wheel.collect_expired(kNever - 1).empty(), "deactivated slots fall silent");
  std::printf("     events collected while debtors renewed: %zu\n", collected);
}

int main() {
  graph_shape();
  dangling_and_generations();
  freshness();
  phantoms_and_hubs();
  escrow();
  guards();
  leases();
  formed_guards();
  revoked_grants();
  concurrency();
  liveness();
  std::printf("%s, IncidenceRow %zu bytes, LeaseSlot %zu bytes\n", failures == 0 ? "all passed" : "FAILURES", sizeof(IncidenceRow), sizeof(LeaseSlot));
  return failures == 0 ? 0 : 1;
}
```

E Kotlin reference engine

This appendix reproduces the Kotlin engine of Section 13.3 in full. The listings are read from the same files that the tests compile, so the code printed here is the code that passed. The engine targets Kotlin 2.4 on the JVM, uses only the standard library, and runs its tests with `gradle test` in `code/kotlin`; all 59 passed in the run of 12 November 2026. Figure 7 shows how the parts fit together, Listing 8 lists the files with the listing that prints each, and Table 10 maps each file to the part of the paper it implements. The engine listings follow in dependency order, and the tests follow the engine.

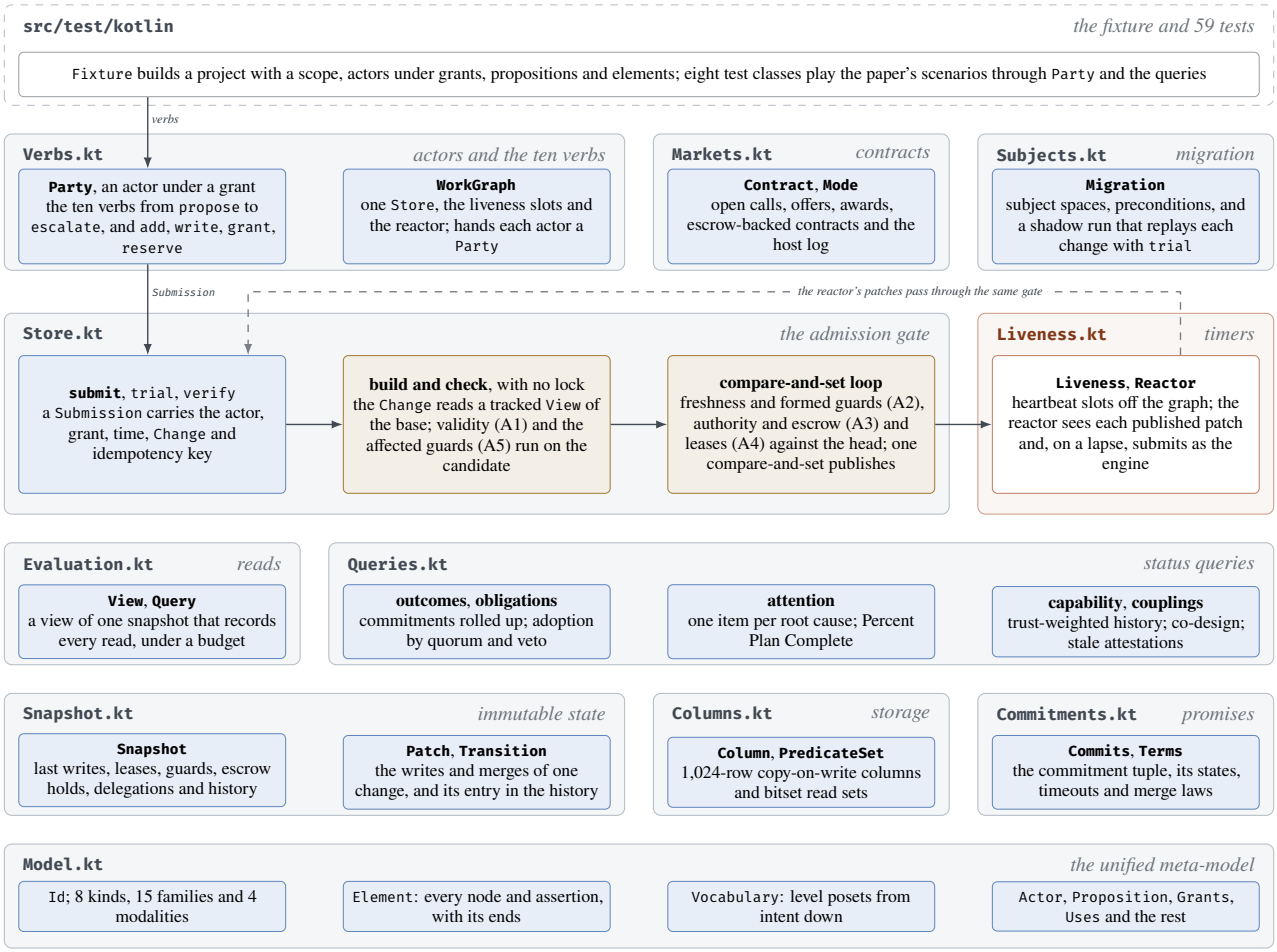

Figure 7: High-level design of the Kotlin engine. Each panel is one file; the bands run from the callers at the top to the meta-model at the bottom. A verb builds a `Change`. The store builds and checks it on the base snapshot without a lock, re-checks against the head what may have changed since, and publishes the result with one compare-and-set. Every published patch reaches the reactor, and the reactor's own patches pass through the same gate (dashed). Amber marks the steps that can refuse a change.

code/kotlin | **Listing 8.** The files of the engine and its tests, in reading order. Files

```
code/kotlin/
├── build.gradle.kts                    Kotlin 2.4, Java 25, JUnit
├── settings.gradle.kts                 project name, plugin sources
└── src/
    ├── main/kotlin/workgraph/
    │   ├── Model.kt           Listing 9   identifiers and the meta-model
    │   ├── Columns.kt         Listing 10  columns and read sets
    │   ├── Evaluation.kt      Listing 11  queries, views and budgets
    │   ├── Commitments.kt     Listing 12  the commitment tuple
    │   ├── Snapshot.kt        Listing 13  snapshots and their indexes
    │   ├── Store.kt           Listing 14  the admission gate
    │   ├── Verbs.kt           Listing 15  the ten verbs and grants
    │   ├── Liveness.kt        Listing 16  heartbeats and the reactor
    │   ├── Queries.kt         Listing 17  status queries and capability
    │   ├── Markets.kt         Listing 18  modes, contracts, host log
    │   └── Subjects.kt        Listing 19  subject spaces and migration
    └── test/kotlin/workgraph/
        ├── Fixture.kt         Listing 20  the shared test project
        ├── AdmissionTest.kt   Listing 21  14 tests: admission
        ├── CommitmentTest.kt  Listing 22  13 tests: commitments
        ├── LivenessTest.kt    Listing 23  7 tests: liveness
        ├── StoreTest.kt       Listing 24  6 tests: escrow and threads
        ├── QueryTest.kt       Listing 25  8 tests: status queries
        ├── MarketTest.kt      Listing 26  4 tests: markets
        ├── MigrationTest.kt   Listing 27  2 tests: migration
        └── ModelTest.kt       Listing 28  5 tests: model and columns
```

File	Lines	What it implements	Paper
`Model.kt`	160	Identifiers, the 8 kinds, 15 families and 4 modalities, level posets, nodes and assertions	Sections 5 and 13.1
`Columns.kt`	64	Persistent 1,024-row copy-on-write columns and bitset read sets	Section 13.2
`Evaluation.kt`	45	Queries, tracked views, budgets and acting for a principal	Sections 5.1 and 5.3
`Commitments.kt`	72	The commitment tuple, its states, flags, escalations and pending amendments	Section 6
`Snapshot.kt`	127	Snapshots: last writes, leases, guard regions, escrow holds, delegations, idempotency keys and history	Sections 7 and 13.2
`Store.kt`	246	The five admission clauses with witnesses, the verification queue and lock-free publication	Section 7
`Verbs.kt`	230	The ten verbs, grants with revocation, and reservations	Sections 5.3 and 6.1
`Liveness.kt`	95	Heartbeats, review windows, deadlines, horizons and the reactor	Section 8
`Queries.kt`	147	Outcomes, obligations, attention with root-cause coalescing, Percent Plan Complete, capability, couplings and staleness	Sections 5.3, 5.4 and 8
`Markets.kt`	99	Modes, open calls, offers and awards, escrow-backed contracts and the federation host log	Sections 9 and 9.2 to 9.4
`Subjects.kt`	47	Subject spaces, preconditions and shadow migration	Sections 10.4 and 10.5

Table 10: Files of the Kotlin engine and the parts of the paper they implement.

How a change is admitted. A party submits a `Change`, a function from a tracked `View` to a `Patch`. `Store.admit` builds the patch on a snapshot, applies it to a candidate snapshot, and runs the subject checks (A1) and the guards of every commitment whose region the patch touches (A5), recording every predicate those queries read. It then loops: it reads the head, refuses on a read written since the base (A2), on a commitment that began guarding the region since the base, on authority and escrow (A3) and on leases (A4), and otherwise builds the next snapshot and publishes it with one compare-and-set. A failed compare-and-set means another change was published first, and the loop checks again against the new head. Listeners, among them the reactor, see each published patch.

E.1 Engine

workgraph/Model.kt | **Listing 9.** Identifiers, the unified meta-model, vocabularies and the elements of the graph. Kotlin 2.4

```kotlin
package workgraph

import kotlin.jvm.JvmInline

@JvmInline
value class Id(val slot: Int) {
    override fun toString() = "#$slot"
}

const val NEVER = Long.MAX_VALUE
```

```
val ROOT = Id(0)

enum class Kind { ACTOR, PROPOSITION, RECORD, RESOURCE, SCOPE, OPERATOR, STATE, TYPE }

enum class Family {
    COMMITS, ATTESTS, DECIDES, GRANTS, CONTRIBUTES, USES,
    BIND, SESSION, CHANNEL, TRANSITION,
    EXPORT, ACCESS, REFERENCES, DERIVES, REFINES,
}

enum class Modality { INTENDED, REALISED, OBSERVED, CLAIMED }

class Vocabulary(val name: String, val levels: List<String>) {
    init {
        require(levels.firstOrNull() == "Intent") { "$name must start at Intent" }
    }

    fun rank(level: String): Int = levels.indexOf(level).also { require(it >= 0) { "$level is not a level of $name" } }
}

val Software = Vocabulary("software", listOf("Intent", "Design", "Program", "IR", "Artifact", "Place", "Machine"))
val Construction = Vocabulary("construction", listOf("Intent", "Spatial", "System", "Element", "Site"))
val Curriculum = Vocabulary("curriculum", listOf("Intent", "Domain", "Resource", "Chapter"))
val Files = Vocabulary("files", listOf("Intent", "Subject"))

enum class Truth { HOLDS, FAILS, UNKNOWN }

sealed interface Element {
    val scope: Id
}

data class Scope(val name: String, val parent: Id?) : Element {
    override val scope: Id get() = parent ?: ROOT
}

enum class ActorRole { PERSON, TEAM, ORGANISATION, AGENT, SERVICE }

data class Actor(val name: String, val role: ActorRole, override val scope: Id) : Element

data class Proposition(val statement: String, override val scope: Id, val weight: Double = 1.0, val query: Query) : Element

data class Record(val digest: String, val source: Id, val at: Long, val content: String, override val scope: Id) : Element

data class Resource(val name: String, val capacity: Long, override val scope: Id) : Element

data class Subject(
    val kind: Kind,
    val name: String,
    val vocabulary: Vocabulary,
    val level: String,
    val extent: Any?,
    override val scope: Id,
    val modality: Modality = Modality.REALISED,
) : Element

data class Structural(val family: Family, val ends: List<Id>, override val scope: Id) : Element

data class Attests(
    val evidence: Id,
    val proposition: Id,
    val verifier: Id,
    val verdict: Truth,
    val about: Set<Id>,
    val atRevision: Long,
    override val scope: Id,
) : Element

enum class Stance { ADOPT, REJECT, VETO, PERMIT }

data class Decides(
    val decider: Id,
    val choice: Id,
    val authority: Id,
    val stance: Stance,
    val considered: List<Id>,
    val party: Id?,
    val supersedes: Id?,
    override val scope: Id,
) : Element

enum class Right { WRITE, DECIDE, ACCEPT, RESERVE, ACT_FOR }

sealed interface DecisionRule {
    data object Authority : DecisionRule
    data class Quorum(val members: Set<Id>, val needed: Int) : DecisionRule
    data class Veto(val holders: Set<Id>) : DecisionRule
}
```

```

data class Grants(
    val grantor: Id,
    val grantee: Id,
    val rights: Set<Right>,
    val over: Id,
    val condition: Id? = null,
    val expiry: Long = NEVER,
    val budget: Map<Id, Long> = emptyMap(),
    val rule: DecisionRule = DecisionRule.Authority,
    val revoked: Boolean = false,
) : Element {
    override val scope: Id get() = over

    fun live(now: Long) = !revoked && now < expiry
}

data class Contributes(val part: Id, val whole: Id, override val scope: Id) : Element

data class Uses(
    val commitment: Id,
    val resource: Id,
    val amount: Long,
    val from: Long,
    val until: Long,
    val authority: Id,
    override val scope: Id,
) : Element

val Element.kind: Kind?
    get() = when (this) {
        is Scope -> Kind.SCOPE
        is Actor -> Kind.ACTOR
        is Proposition -> Kind.PROPOSITION
        is Record -> Kind.RECORD
        is Resource -> Kind.RESOURCE
        is Subject -> kind
        else -> null
    }

val Element.family: Family?
    get() = when (this) {
        is Commits -> Family.COMMITS
        is Attests -> Family.ATTESTS
        is Decides -> Family.DECIDES
        is Grants -> Family.GRANTS
        is Contributes -> Family.CONTRIBUTES
        is Uses -> Family.USES
        is Structural -> family
        else -> null
    }

fun Element.ends(): List<Id> = when (this) {
    is Commits -> listOfNotNull(terms.debtor, terms.creditor, terms.consequent, terms.antecedent, terms.acceptanceQuery,
        authority) + terms.lease
    is Attests -> listOf(evidence, proposition, verifier) + about
    is Decides -> listOfNotNull(decider, choice, authority, party, supersedes) + considered
    is Grants -> listOfNotNull(grantor, grantee, over, condition) + budget.keys
    is Contributes -> listOf(part, whole)
    is Uses -> listOf(commitment, resource, authority)
    is Structural -> ends
    is Scope -> listOfNotNull(parent)
    else -> emptyList()
}
```

workgraph/Columns.kt | **Listing 10.** Persistent copy-on-write columns and bitset read sets. Kotlin 2.4

```
package workgraph

private const val SHIFT = 10
private const val CHUNK = 1 shl SHIFT
private const val MASK = CHUNK - 1

class Column<T : Any> private constructor(private val chunks: Array<Array<Any?>?>) {
    constructor() : this(arrayOfNulls(0))

    @Suppress("UNCHECKED_CAST")
    operator fun get(id: Id): T? = chunks.getOrNull(id.slot ushr SHIFT)?.get(id.slot and MASK) as T?

    fun with(updates: Map<Id, T?>): Column<T> {
        if (updates.isEmpty()) return this
        val table = chunks.copyOf(maxOf(chunks.size, (updates.keys.maxOf { it.slot } ushr SHIFT) + 1))
        val copied = HashSet<Int>()
        for ((id, value) in updates) {
```

```
            val index = id.slot ushr SHIFT
            if (copied.add(index)) table[index] = table[index]?.copyOf() ?: arrayOfNulls(CHUNK)
            table[index]!![id.slot and MASK] = value
        }
        return Column(table)
    }

    @Suppress("UNCHECKED_CAST")
    fun entries(): Sequence<Pair<Id, T>> = sequence {
        for ((index, chunk) in chunks.withIndex()) {
            if (chunk == null) continue
            for ((offset, value) in chunk.withIndex()) if (value != null) yield(Id((index shl SHIFT) or offset) to value as T)
        }
    }
}

class PredicateSet {
    private var words = LongArray(4)

    fun add(id: Id) {
        val word = id.slot ushr 6
        if (word >= words.size) words = words.copyOf(maxOf(word + 1, words.size * 2))
        words[word] = words[word] or (1L shl id.slot)
    }

    operator fun contains(id: Id): Boolean {
        val word = id.slot ushr 6
        return word < words.size && words[word] and (1L shl id.slot) != 0L
    }

    fun addAll(other: PredicateSet) {
        if (other.words.size > words.size) words = words.copyOf(other.words.size)
        for (index in other.words.indices) words[index] = words[index] or other.words[index]
    }

    fun ids(): Sequence<Id> = sequence {
        for ((word, value) in words.withIndex()) {
            var bits = value
            while (bits != 0L) {
                yield(Id(word * 64 + bits.countTrailingZeroBits()))
                bits = bits and (bits - 1)
            }
        }
    }

    fun toSet(): Set<Id> = ids().toSet()
}
```

workgraph/Evaluation.kt | **Listing 11.** Queries, tracked views, budgets and acting for a principal. Kotlin 2.4

```
package workgraph

fun interface Query {
    fun evaluate(view: View): Truth
}

class BudgetExceeded : RuntimeException("the query exceeded its synchronous budget")

class View(val snapshot: Snapshot, val now: Long, private val budget: Int = Int.MAX_VALUE) {
    val reads = PredicateSet()
    private var steps = 0

    fun read(id: Id): Element? {
        if (++steps > budget) throw BudgetExceeded()
        reads.add(id)
        return snapshot[id]
    }

    inline fun <reified T : Element> get(id: Id): T =
        read(id) as? T ?: refuse(id, "$id is not a ${T::class.simpleName}")

    fun truth(proposition: Id): Truth = get<Proposition>(proposition).query.evaluate(this)

    fun holds(proposition: Id): Boolean = truth(proposition) == Truth.HOLDS

    fun actsFor(actor: Id, principal: Id?): Boolean =
        principal != null && (actor == principal || snapshot.delegationsOf(principal).any { id ->
            val grant = read(id) as? Grants
            grant != null && grant.grantee == actor && Right.ACT_FOR in grant.rights && grant.live(now)
        })
}

class Evaluated(val truth: Truth?, val reads: PredicateSet)
```

```
fun Snapshot.evaluate(proposition: Id, now: Long, budget: Int = Int.MAX_VALUE): Evaluated {
    val view = View(this, now, budget)
    val truth = try {
        view.truth(proposition)
    } catch (_: BudgetExceeded) {
        null
    }
    return Evaluated(truth, view.reads)
}

fun refuse(predicate: Id?, reason: String): Nothing = throw Refusal(Witness(Clause.VALIDITY, predicate, reason = reason))
```

workgraph/Commitments.kt | **Listing 12.** The commitment tuple, its states and its terms. Kotlin 2.4

```
package workgraph

enum class Temporal { ACHIEVE, MAINTAIN }

enum class State { PROPOSED, CONDITIONAL, ACTIVE, DISCHARGED, VIOLATED, RELEASED, CANCELLED, DECLINED }

enum class Timeout { FAIL_CLOSED, OPTIMISTIC_AT_RISK }

enum class MergeLaw { EXCLUSIVE, COMMUTATIVE }

val LIVE = setOf(State.CONDITIONAL, State.ACTIVE)
val OPEN = setOf(State.PROPOSED, State.CONDITIONAL, State.ACTIVE)
val WITHDRAWN = setOf(State.DECLINED, State.CANCELLED, State.RELEASED)

data class Terms(
    val debtor: Id?,
    val creditor: Id,
    val consequent: Id,
    val antecedent: Id? = null,
    val acceptance: Id? = null,
    val lease: Set<Id> = emptySet(),
    val temporal: Temporal = Temporal.ACHIEVE,
    val deadline: Long = NEVER,
    val heartbeat: Long = NEVER,
    val review: Long = 0,
    val creditorDecides: Boolean = false,
    val deemedAccepted: Boolean = false,
    val timeout: Timeout = Timeout.FAIL_CLOSED,
    val merge: MergeLaw = MergeLaw.EXCLUSIVE,
    val arbiter: Id? = null,
    val category: String = "",
) {
    val acceptanceQuery: Id get() = acceptance ?: consequent
}

data class Escalation(val to: Set<Id>, val witness: String, val at: Long)

data class Amendment(val terms: Terms, val consents: Set<Id>)

data class Commits(
    val terms: Terms,
    val authority: Id,
    val proposer: Id,
    override val scope: Id,
    val state: State = State.PROPOSED,
    val consented: Boolean = proposer == terms.creditor,
    val atRisk: Boolean = false,
    val inReview: Boolean = false,
    val reviewSince: Long? = null,
    val parent: Id? = null,
    val respondsTo: Id? = null,
    val delegates: Set<Id> = emptySet(),
    val escalation: Escalation? = null,
    val pending: Amendment? = null,
    val closedAt: Long? = null,
    val reason: String? = null,
) : Element {
    val debtor: Id? get() = terms.debtor
    val creditor: Id get() = terms.creditor
    val holdsLease: Boolean get() = state == State.ACTIVE && parent == null

    fun close(state: State, now: Long, reason: String? = null) =
        copy(state = state, closedAt = now, atRisk = false, inReview = false, reviewSince = null, reason = reason)

    fun escalate(witness: String, now: Long, to: Set<Id>) = copy(escalation = Escalation(to, witness, now))

    fun resumeAfterReview(now: Long): Commits {
        val paused = now - (reviewSince ?: now)
        val deadline = if (terms.deadline == NEVER) NEVER else terms.deadline + paused
        return copy(terms = terms.copy(deadline = deadline), inReview = false, reviewSince = null)
```

```
    }
}
```

workgraph/Snapshot.kt | **Listing 13.** Snapshots and the indexes kept with them. Kotlin 2.4

```
package workgraph

data class Transition(val revision: Long, val verb: String, val actor: Id, val commitment: Id?, val cites: List<Id>, val at: Long)

class Patch(
    val verb: String,
    val writes: Map<Id, Element> = emptyMap(),
    val merges: Map<Id, (Element) -> Element> = emptyMap(),
    val commitment: Id? = null,
    val cites: List<Id> = emptyList(),
    val exempt: Set<Id> = emptySet(),
    val permissions: Set<Id> = emptySet(),
    val observation: Boolean = false,
) {
    val footprint: Set<Id> get() = writes.keys + merges.keys

    fun with(extra: Map<Id, Element>) = Patch(verb, writes + extra, merges, commitment, cites, exempt, permissions, observation)
}

class Snapshot private constructor(
    val revision: Long,
    private val elements: Column<Element>,
    private val written: Column<Long>,
    private val leases: Column<Id>,
    private val guards: Column<Set<Id>>,
    private val observed: Column<Set<Id>>,
    private val holds: Column<List<Id>>,
    private val delegations: Column<List<Id>>,
    private val keys: Column<Long>,
    private val history: Column<Transition>,
) {
    operator fun get(id: Id): Element? = elements[id]
    fun lastWrite(id: Id): Long = written[id] ?: 0L
    fun leaseHolder(predicate: Id): Id? = leases[predicate]
    fun guardsOf(predicate: Id): Set<Id> = guards[predicate].orEmpty()
    fun observedBy(commitment: Id): Set<Id> = observed[commitment].orEmpty()
    fun holdsOn(resource: Id): List<Id> = holds[resource].orEmpty()
    fun delegationsOf(principal: Id): List<Id> = delegations[principal].orEmpty()
    fun admitted(key: Id): Long? = keys[key]
    fun elements(): Sequence<Pair<Id, Element>> = elements.entries()
    fun transitions(): Sequence<Transition> = history.entries().map { it.second }

    inline fun <reified T : Element> all(): Sequence<Pair<Id, T>> =
        elements().mapNotNull { (id, element) -> (element as? T)?.let { id to it } }

    internal fun next(
        patch: Patch,
        revision: Long,
        actor: Id,
        now: Long,
        observedNow: Map<Id, Set<Id>> = emptyMap(),
        key: Id? = null,
    ): Snapshot {
        val updates = LinkedHashMap<Id, Element>(patch.writes)
        for ((id, merge) in patch.merges) updates[id] = merge(updates[id] ?: get(id) ?: refuse(id, "nothing to merge into"))

        val leaseUpdates = HashMap<Id, Id?>()
        for (id in updates.keys) {
            val old = get(id) as? Commits ?: continue
            if (old.holdsLease) for (p in old.terms.lease) if (leases[p] == id) leaseUpdates[p] = null
        }
        for ((id, new) in updates) if (new is Commits && new.holdsLease) for (p in new.terms.lease) leaseUpdates[p] = id

        val regions = HashMap<Id, MutableSet<Id>>()
        val observedUpdates = HashMap<Id, Set<Id>?>()
        for (id in updates.filterValues { it is Commits }.keys + observedNow.keys) {
            val old = get(id) as? Commits
            val new = (updates[id] ?: get(id)) as Commits
            val oldRegion = if (old != null && old.state in LIVE) old.terms.lease + observedBy(id) else emptySet()
            val newObserved = if (new.state in LIVE) observedNow[id] ?: observedBy(id) else emptySet()
            val newRegion = if (new.state in LIVE) new.terms.lease + newObserved else emptySet()
            observedUpdates[id] = newObserved.ifEmpty { null }
            for (p in oldRegion - newRegion) regions.getOrPut(p) { guardsOf(p).toMutableSet() }.remove(id)
            for (p in newRegion - oldRegion) regions.getOrPut(p) { guardsOf(p).toMutableSet() }.add(id)
        }

        val holdUpdates = HashMap<Id, List<Id>>()
```

```
        val delegationUpdates = HashMap<Id, List<Id>>()
        for ((id, new) in updates) {
            if (get(id) != null) continue
            if (new is Uses) holdUpdates[new.resource] = (holdUpdates[new.resource] ?: holdsOn(new.resource)) + id
            if (new is Grants && Right.ACT_FOR in new.rights) {
                delegationUpdates[new.grantor] = (delegationUpdates[new.grantor] ?: delegationsOf(new.grantor)) + id
            }
        }

        return Snapshot(
            revision,
            elements.with(updates),
            written.with(updates.keys.associateWith { revision }),
            leases.with(leaseUpdates),
            guards.with(regions.mapValues { (_, set) -> if (set.isEmpty()) null else set.toSet() }),
            observed.with(observedUpdates),
            holds.with(holdUpdates),
            delegations.with(delegationUpdates),
            if (key == null) keys else keys.with(mapOf(key to revision)),
            history.with(mapOf(Id(revision.toInt()) to Transition(revision, patch.verb, actor, patch.commitment, patch.cites,
                now))),
        )
    }

    internal fun inherit(from: Snapshot) =
        Snapshot(revision, elements, written, leases, guards, observed, holds, delegations, from.keys,
            from.history.with(history.entries().toMap()))

    companion object {
        val EMPTY = Snapshot(0, Column(), Column(), Column(), Column(), Column(), Column(), Column(), Column(), Column())
    }
}

fun Snapshot.liveHolds(resource: Id, now: Long): List<Uses> =
    holdsOn(resource).mapNotNull { get(it) as? Uses }.filter { use ->
        now < use.until && (get(use.commitment) as? Commits)?.state.let { it == null || it in OPEN }
    }

fun Snapshot.available(resource: Id, from: Long, until: Long, now: Long): Long =
    ((get(resource) as? Resource)?.capacity ?: 0L) -
        liveHolds(resource, now).filter { it.from < until && from < it.until }.sumOf { it.amount }

fun Snapshot.spent(grant: Id, resource: Id): Long =
    holdsOn(resource).mapNotNull { get(it) as? Uses }
        .filter { use -> use.authority == grant && (get(use.commitment) as? Commits)?.state.let { it == null || it !in
            WITHDRAWN } }
        .sumOf { it.amount }

tailrec fun Snapshot.within(scope: Id, ancestor: Id): Boolean {
    if (scope == ancestor) return true
    val parent = (get(scope) as? Scope)?.parent ?: return false
    return within(parent, ancestor)
}
```

workgraph/Store.kt | **Listing 14.** The admission gate: five clauses, witnesses and lock-free publication. Kotlin 2.4

```
package workgraph

import kotlin.concurrent.atomics.AtomicInt
import kotlin.concurrent.atomics.AtomicReference

enum class Clause { VALIDITY, FRESHNESS, AUTHORITY, LEASE, GUARD }

data class Witness(
    val clause: Clause,
    val predicate: Id? = null,
    val commitment: Id? = null,
    val holder: Id? = null,
    val creditor: Id? = null,
    val reason: String = "",
)

class Refusal(val witness: Witness) : RuntimeException(witness.reason)

sealed interface Outcome {
    data class Admitted(val revision: Long) : Outcome
    data class Refused(val witness: Witness) : Outcome
}

fun interface Change {
    fun build(view: View): Patch
}

class Check(val name: String, val query: Query)
```

```
class Submission(
    val actor: Id,
    val grant: Id,
    val now: Long,
    val change: Change,
    val key: Id? = null,
    val base: Snapshot? = null,
    val budget: Int? = null,
)

sealed interface Event {
    data class AtRisk(val commitment: Id, val debtor: Id?, val creditor: Id) : Event
}

fun interface Listener {
    fun committed(snapshot: Snapshot, patch: Patch, events: List<Event>, now: Long)
}

private class Guarded(val evaluated: Set<Id>, val observed: Map<Id, Set<Id>>, val atRisk: Map<Id, Element>, val events:
    List<Event>)

internal inline fun <T> AtomicReference<T>.update(change: (T) -> T) {
    while (true) {
        val old = load()
        if (compareAndSet(old, change(old))) return
    }
}

class Store(val checks: List<Check> = emptyList(), val budget: Int = 10_000, initial: Snapshot = genesis()) {
    private val head = AtomicReference(initial)
    private val slots = AtomicInt(initial.elements().maxOf { it.first.slot } + 1)
    private val listeners = AtomicReference(emptyList<Listener>())
    private val waiting = AtomicReference(emptyList<Submission>())

    fun snapshot(): Snapshot = head.load()
    fun allocate(): Id = Id(slots.fetchAndAdd(1))
    fun listen(listener: Listener) = listeners.update { it + listener }
    fun forget(listener: Listener) = listeners.update { it - listener }
    fun pending(): List<Submission> = waiting.load()

    fun watch(predicates: Set<Id>, wake: (Snapshot, Set<Id>) -> Unit): Listener =
        Listener { snapshot, patch, _, _ ->
            val touched = patch.footprint intersect predicates
            if (touched.isNotEmpty()) wake(snapshot, touched)
        }.also(::listen)

    fun submit(submission: Submission): Outcome = admit(submission, publish = true)

    fun trial(submission: Submission): Outcome = admit(submission, publish = false)

    fun verify(budget: Int = Int.MAX_VALUE): List<Outcome> =
        waiting.exchange(emptyList()).map { submit(Submission(it.actor, it.grant, it.now, it.change, it.key, budget = budget))
            }

    private fun admit(sub: Submission, publish: Boolean): Outcome {
        sub.key?.let { key -> head.load().admitted(key)?.let { return Outcome.Admitted(it) } }
        val base = sub.base ?: head.load()
        val view = View(base, sub.now)
        val patch: Patch
        val guarded: Guarded
        try {
            patch = sub.change.build(view)
            val candidate = base.next(patch, base.revision + 1, sub.actor, sub.now)
            validity(patch, candidate, view, sub.now)
            guarded = guard(sub, patch, base, candidate, view, publish)
        } catch (refusal: Refusal) {
            return Outcome.Refused(refusal.witness)
        }
        val created = patch.writes.keys.filterTo(HashSet()) { base[it] == null }
        while (true) {
            val current = head.load()
            sub.key?.let { key -> current.admitted(key)?.let { return Outcome.Admitted(it) } }
            val refusal = freshness(view, patch, base, current, guarded, created)
                ?: authority(sub, patch, current)
                ?: lease(sub, patch, current)
            if (refusal != null) return Outcome.Refused(refusal)
            val effective = patch.with(guarded.atRisk)
            val next = current.next(effective, current.revision + 1, sub.actor, sub.now, guarded.observed, sub.key)
            if (!publish) return Outcome.Admitted(next.revision)
            if (head.compareAndSet(current, next)) {
                listeners.load().forEach { it.committed(next, effective, guarded.events, sub.now) }
                return Outcome.Admitted(next.revision)
            }
        }
    }

```

```
    private fun validity(patch: Patch, candidate: Snapshot, view: View, now: Long) {
        for (check in checks) {
            val after = View(candidate, now)
            if (check.query.evaluate(after) != Truth.HOLDS) {
                throw Refusal(Witness(Clause.VALIDITY, after.reads.ids().firstOrNull { it in patch.footprint }, reason = "check
                    ${check.name} fails"))
            }
            view.reads.addAll(after.reads)
        }
    }

    private fun guard(sub: Submission, patch: Patch, base: Snapshot, candidate: Snapshot, view: View, publish: Boolean):
        Guarded {
        val budget = sub.budget ?: budget
        val affected = patch.footprint.flatMapTo(HashSet()) { base.guardsOf(it) } - patch.exempt
        val observed = HashMap<Id, Set<Id>>()
        val atRisk = HashMap<Id, Element>()
        val events = ArrayList<Event>()
        for (id in affected) {
            val c = view.get<Commits>(id)
            if (c.state !in LIVE || id in patch.writes) continue
            val reads = PredicateSet()
            var complete = true
            for (condition in listOfNotNull(c.terms.acceptanceQuery, c.terms.antecedent)) {
                val before = base.evaluate(condition, sub.now, budget)
                val after = candidate.evaluate(condition, sub.now, budget)
                view.reads.addAll(before.reads)
                view.reads.addAll(after.reads)
                reads.addAll(after.reads)
                if (before.truth == null || after.truth == null) {
                    complete = false
                    if (c.terms.timeout == Timeout.FAIL_CLOSED) {
                        if (publish) waiting.update { it + sub }
                        throw Refusal(Witness(Clause.GUARD, condition, id, c.debtor, c.creditor, "$id needs asynchronous
                            verification"))
                    }
                    atRisk[id] = c.copy(atRisk = true)
                    events += Event.AtRisk(id, c.debtor, c.creditor)
                } else if (before.truth == Truth.HOLDS && after.truth == Truth.FAILS && !patch.observation) {
                    val predicate = after.reads.ids().firstOrNull { it in patch.footprint }
                    throw Refusal(Witness(Clause.GUARD, predicate, id, c.debtor, c.creditor, "the change breaks $condition of
                        $id"))
                }
            }
            if (complete) observed[id] = reads.toSet()
        }
        for ((id, element) in patch.writes) {
            if (element !is Commits || element.state !in LIVE) continue
            val reads = PredicateSet()
            for (condition in listOfNotNull(element.terms.acceptanceQuery, element.terms.antecedent))
                reads.addAll(candidate.evaluate(condition, sub.now).reads)
            observed[id] = reads.toSet()
        }
        return Guarded(affected, observed, atRisk, events)
    }

    private fun freshness(view: View, patch: Patch, base: Snapshot, current: Snapshot, guarded: Guarded, created: Set<Id>):
        Witness? {
        if (current.revision == base.revision) return null
        view.reads.ids().firstOrNull { current.lastWrite(it) > base.revision }?.let {
            return Witness(Clause.FRESHNESS, it, reason = "$it was written after revision ${base.revision}")
        }
        val formed = patch.footprint.asSequence().flatMap { current.guardsOf(it) }
            .firstOrNull { it !in guarded.evaluated && it !in patch.exempt && it !in created }
        return formed?.let { Witness(Clause.FRESHNESS, commitment = it, reason = "$it began guarding this region after revision
            ${base.revision}") }
    }

    private fun authority(sub: Submission, patch: Patch, at: Snapshot): Witness? {
        val grant = at[sub.grant] as? Grants ?: return Witness(Clause.AUTHORITY, sub.grant, reason = "${sub.grant} is not a
            grant")
        fun refused(reason: String, predicate: Id? = sub.grant) = Witness(Clause.AUTHORITY, predicate, holder = grant.grantor,
            reason = reason)
        if (grant.grantee != sub.actor) return refused("${sub.grant} belongs to ${grant.grantee}")
        if (!grant.live(sub.now)) return refused("${sub.grant} is revoked or expired")
        grant.condition?.let { if (at.evaluate(it, sub.now, budget).truth != Truth.HOLDS) return refused("the credential behind
            ${sub.grant} does not hold", it) }
        for (id in patch.footprint) {
            val element = patch.writes[id] ?: at[id] ?: continue
            if (!at.within(element.scope, grant.over)) return refused("$id lies outside ${grant.over}", id)
            if ((element is Subject || element is Structural) && Right.WRITE !in grant.rights) return refused("${sub.grant}
                cannot write the subject", id)
        }
        val demand = patch.writes.filter { (id, element) -> element is Uses && at[id] == null }.values.filterIsInstance<Uses>()
        for ((resource, uses) in demand.groupBy { it.resource }) {
            val wanted = uses.sumOf { it.amount }
            if (Right.RESERVE !in grant.rights) return refused("${sub.grant} cannot reserve", resource)
```

```
            if (at.available(resource, uses.minOf { it.from }, uses.maxOf { it.until }, sub.now) < wanted) return refused("$resource lacks $wanted free units", resource)
            if (at.spent(sub.grant, resource) + wanted > (grant.budget[resource] ?: 0L)) return refused("${sub.grant} has no budget left on $resource", resource)
        }
        return null
    }

    private fun lease(sub: Submission, patch: Patch, at: Snapshot): Witness? {
        val view = View(at, sub.now)
        for (id in patch.footprint) {
            val holder = at.leaseHolder(id) ?: continue
            if (holder in patch.exempt) continue
            val c = at[holder] as? Commits ?: continue
            val allowed = view.actsFor(sub.actor, c.debtor) ||
                c.delegates.any { child ->
                    val delegated = at[child] as? Commits
                    delegated != null && delegated.state == State.ACTIVE && id in delegated.terms.lease &&
                        view.actsFor(sub.actor, delegated.debtor)
                } ||
                (id !in patch.writes && c.terms.merge == MergeLaw.COMMUTATIVE) ||
                patch.permissions.any { permission ->
                    val decision = at[permission] as? Decides
                    decision != null && decision.stance == Stance.PERMIT && decision.choice == holder &&
                        decision.party == sub.actor && view.actsFor(decision.decider, c.creditor)
                }
            if (!allowed) return Witness(Clause.LEASE, id, holder, c.debtor, c.creditor, "$id is leased to ${c.debtor} under $holder")
        }
        for ((id, element) in patch.writes) {
            if (element !is Commits || !element.holdsLease) continue
            val held = (at[id] as? Commits)?.takeIf { it.holdsLease }?.terms?.lease.orEmpty()
            for (predicate in element.terms.lease - held) {
                val other = at.leaseHolder(predicate) ?: continue
                if (other == id) continue
                val holder = at[other] as? Commits
                return Witness(Clause.LEASE, predicate, other, holder?.debtor, holder?.creditor, "$predicate is already leased under $other")
            }
        }
        return null
    }

    companion object {
        val ENGINE = Id(1)
        val ENGINE_GRANT = Id(2)

        fun genesis(): Snapshot = Snapshot.EMPTY.next(
            Patch(
                "genesis",
                mapOf(
                    ROOT to Scope("root", null),
                    ENGINE to Actor("engine", ActorRole.SERVICE, ROOT),
                    ENGINE_GRANT to Grants(ENGINE, ENGINE, Right.entries.toSet(), ROOT),
                ),
            ),
            0,
            ENGINE,
            0,
        )
    }
}
```

workgraph/Verbs.kt | **Listing 15.** The ten verbs, grants and reservations. Kotlin 2.4

```
package workgraph

class WorkGraph(val store: Store = Store()) {
    val liveness = Liveness()
    val reactor = Reactor(this).also(store::listen)

    fun party(actor: Id, grant: Id) = Party(this, actor, grant)

    val engine: Party get() = party(Store.ENGINE, Store.ENGINE_GRANT)
}

class Party(val graph: WorkGraph, val actor: Id, val grant: Id) {
    private val store: Store get() = graph.store

    fun submit(now: Long, key: Id? = null, change: Change): Outcome = store.submit(Submission(actor, grant, now, change, key))

    private fun creating(now: Long, build: (Id, View) -> Patch): Pair<Id, Outcome> {
        val id = store.allocate()
        return id to submit(now, key = id) { view -> build(id, view) }
```

```
    }

    fun add(element: Element, now: Long) = creating(now) { id, _ -> Patch("add", mapOf(id to element)) }

    fun write(id: Id, now: Long, permissions: Set<Id> = emptySet(), edit: (Element) -> Element) = submit(now) { view ->
        Patch("write", mapOf(id to edit(view.read(id) ?: refuse(id, "$id does not exist"))), permissions = permissions)
    }

    fun merge(id: Id, now: Long, edit: (Element) -> Element) = submit(now) { Patch("merge", merges = mapOf(id to edit)) }

    fun observe(id: Id, element: Element, now: Long) = submit(now) { Patch("observe", mapOf(id to element), observation = true)
        }

    fun grant(
        to: Id,
        rights: Set<Right>,
        over: Id,
        now: Long,
        condition: Id? = null,
        expiry: Long = NEVER,
        budget: Map<Id, Long> = emptyMap(),
        rule: DecisionRule = DecisionRule.Authority,
    ) = creating(now) { id, view ->
        val mine = view.get<Grants>(grant)
        if (!mine.rights.containsAll(rights)) refuse(grant, "a grant cannot pass on rights it lacks")
        if (!view.snapshot.within(over, mine.over)) refuse(over, "a grant cannot reach beyond its own scope")
        if (expiry > mine.expiry) refuse(grant, "a grant cannot outlive its source")
        Patch("grant", mapOf(id to Grants(actor, to, rights, over, condition, expiry, budget, rule)))
    }

    fun revoke(target: Id, now: Long) = submit(now) { view ->
        val granted = view.get<Grants>(target)
        if (!view.actsFor(actor, granted.grantor)) refuse(target, "only the grantor revokes")
        Patch("revoke", mapOf(target to granted.copy(revoked = true)))
    }

    fun propose(terms: Terms, now: Long, respondsTo: Id? = null) = creating(now) { id, view ->
        val authority = view.get<Grants>(grant)
        if (terms.debtor == null && actor != terms.creditor) refuse(id, "only the creditor can post an open call")
        respondsTo?.let { call ->
            val open = view.get<Commits>(call)
            if (open.state != State.PROPOSED || open.debtor != null) refuse(call, "$call is not an open call")
            if (terms.creditor != open.creditor) refuse(call, "an offer answers the creditor of the call")
        }
        Patch("propose", mapOf(id to Commits(terms, grant, actor, authority.over, respondsTo = respondsTo)), commitment = id)
    }

    fun commit(c: Id, now: Long) = submit(now) { view ->
        val x = view.get<Commits>(c)
        val debtor = x.debtor ?: refuse(c, "$c has no debtor yet")
        if (x.state != State.PROPOSED) refuse(c, "$c is ${x.state}")
        if (!view.actsFor(actor, debtor)) refuse(c, "only the debtor commits")
        if (!x.consented) refuse(c, "the creditor has not consented to $c")
        val ready = x.terms.antecedent?.let(view::holds) ?: true
        Patch("commit", mapOf(c to x.copy(authority = grant, state = if (ready) State.ACTIVE else State.CONDITIONAL)),
            commitment = c)
    }

    fun decline(c: Id, reason: String, now: Long) = submit(now) { view ->
        val x = view.get<Commits>(c)
        if (x.state != State.PROPOSED) refuse(c, "only a proposal can be declined")
        if (!view.actsFor(actor, x.debtor) && !view.actsFor(actor, x.creditor)) refuse(c, "only a party declines")
        Patch("decline", mapOf(c to x.close(State.DECLINED, now, reason)), commitment = c)
    }

    fun delegate(c: Id, terms: Terms, now: Long): Pair<Id, Outcome> {
        val link = store.allocate()
        return creating(now) { child, view ->
            val parent = view.get<Commits>(c)
            if (parent.state !in LIVE) refuse(c, "only a live commitment delegates")
            if (!view.actsFor(actor, parent.debtor)) refuse(c, "only the debtor delegates")
            if (terms.creditor != parent.debtor) refuse(c, "the debtor of $c is the creditor of its delegation")
            if (!parent.terms.lease.containsAll(terms.lease)) refuse(c, "a delegated lease stays inside the parent's")
            Patch(
                "delegate",
                mapOf(
                    child to Commits(terms, grant, actor, parent.scope, consented = true, parent = c),
                    link to Contributes(terms.consequent, parent.terms.consequent, parent.scope),
                    c to parent.copy(delegates = parent.delegates + child),
                ),
                commitment = c,
            )
        }
    }

    fun report(c: Id, content: String, now: Long, about: Set<Id> = emptySet()): Outcome {
        val attestation = store.allocate()
```

```
        val (_, outcome) = creating(now) { record, view ->
            val x = view.get<Commits>(c)
            if (x.state != State.ACTIVE) refuse(c, "only an active commitment takes reports")
            val delegated = x.delegates.any { child ->
                (view.read(child) as? Commits)?.let { it.state == State.ACTIVE && view.actsFor(actor, it.debtor) } == true
            }
            if (!view.actsFor(actor, x.debtor) && !delegated) refuse(c, "only the debtor or a delegate reports")
            val verdict = view.truth(x.terms.acceptanceQuery)
            val timely = now <= x.terms.deadline
            val achieve = x.terms.temporal == Temporal.ACHIEVE
            val updated = when {
                achieve && !x.terms.creditorDecides && verdict == Truth.HOLDS && timely -> x.close(State.DISCHARGED, now)
                achieve && x.terms.creditorDecides && timely && !x.inReview -> x.copy(inReview = true, reviewSince = now,
                    atRisk = false)
                else -> x.copy(atRisk = false)
            }
            val writes = mutableMapOf<Id, Element>(
                record to Record(fnv(content), actor, now, content, x.scope),
                attestation to Attests(record, x.terms.consequent, actor, verdict, about, view.snapshot.revision, x.scope),
            )
            if (updated != x) writes[c] = updated
            Patch("report", writes, commitment = c, cites = listOf(record))
        }
        if (outcome is Outcome.Admitted) graph.liveness.renew(c, now)
        return outcome
    }

    fun decide(
        choice: Id,
        stance: Stance,
        now: Long,
        considered: List<Id> = emptyList(),
        party: Id? = null,
        supersedes: Id? = null,
    ) = creating(now) { id, view ->
        val authority = view.get<Grants>(grant)
        val writes = mutableMapOf<Id, Element>(id to Decides(actor, choice, grant, stance, considered, party, supersedes,
            authority.over))
        when (val target = view.read(choice)) {
            is Commits -> writes += settle(view, authority, choice, target, stance, considered, now)
            else -> if (Right.DECIDE !in authority.rights) refuse(choice, "$grant carries no right to decide")
        }
        supersedes?.let { old ->
            val previous = view.get<Decides>(old)
            if (previous.choice != choice) refuse(old, "$old decided something else")
            if (!view.actsFor(actor, previous.decider)) refuse(old, "only ${previous.decider} supersedes $old")
        }
        Patch("decide", writes, commitment = choice.takeIf { view.snapshot[it] is Commits }, cites = considered)
    }

    private fun settle(view: View, authority: Grants, c: Id, x: Commits, stance: Stance, considered: List<Id>, now: Long):
        Map<Id, Element> {
        val creditor = view.actsFor(actor, x.creditor)
        val arbiter = x.terms.arbiter?.let { view.actsFor(actor, it) && Right.ACCEPT in authority.rights } == true
        val reviewing = x.state == State.ACTIVE && (x.inReview || x.terms.creditorDecides) && (creditor || arbiter)
        return when {
            stance == Stance.PERMIT && creditor -> emptyMap()
            x.state == State.PROPOSED && stance == Stance.ADOPT && creditor -> award(view, c, x, considered, now)
            x.state == State.PROPOSED && stance == Stance.REJECT && creditor -> mapOf(c to x.close(State.DECLINED, now,
                "rejected by the creditor"))
            reviewing && stance == Stance.ADOPT -> {
                if (!view.holds(x.terms.acceptanceQuery)) refuse(x.terms.acceptanceQuery, "the acceptance query of $c does not
                    hold")
                mapOf(c to x.close(State.DISCHARGED, now))
            }
            reviewing && stance == Stance.REJECT && x.inReview -> {
                if (considered.isEmpty()) refuse(c, "a rejection cites the defects it found")
                mapOf(c to x.resumeAfterReview(now))
            }
            else -> refuse(c, "$actor cannot $stance $c while it is ${x.state}")
        }
    }

    private fun award(view: View, c: Id, x: Commits, considered: List<Id>, now: Long): Map<Id, Element> {
        val writes = mutableMapOf<Id, Element>(c to x.copy(consented = true))
        val call = x.respondsTo ?: return writes
        writes[call] = view.get<Commits>(call).close(State.DECLINED, now, "awarded to $c")
        for (offer in considered - c) {
            val other = view.get<Commits>(offer)
            if (other.respondsTo == call && other.state == State.PROPOSED) writes[offer] = other.close(State.DECLINED, now,
                "another offer was selected")
        }
        return writes
    }

    fun amend(c: Id, terms: Terms, now: Long) = submit(now) { view ->
        val x = view.get<Commits>(c)
```

```
        if (x.state !in setOf(State.CONDITIONAL, State.ACTIVE, State.VIOLATED)) refuse(c, "$c cannot be amended while ${x.state}")
        val parties = setOfNotNull(x.creditor, x.debtor, terms.debtor)
        val mine = parties.filterTo(HashSet()) { view.actsFor(actor, it) }
        if (mine.isEmpty()) refuse(c, "only a party amends $c")
        val consents = x.pending?.takeIf { it.terms == terms }?.consents.orEmpty() + mine
        if (!consents.containsAll(parties)) {
            Patch("amend", mapOf(c to x.copy(pending = Amendment(terms, consents))), commitment = c)
        } else {
            val state = if (x.state == State.VIOLATED) State.ACTIVE else x.state
            val agreed = x.copy(terms = terms, pending = null, state = state, atRisk = false, escalation = null, closedAt = null)
            Patch("amend", mapOf(c to agreed), commitment = c, exempt = setOf(c))
        }
    }

    fun release(c: Id, now: Long) = submit(now) { view ->
        val x = view.get<Commits>(c)
        if (x.state !in OPEN && x.state != State.VIOLATED) refuse(c, "$c is already closed")
        if (!view.actsFor(actor, x.creditor)) refuse(c, "only the creditor releases")
        val unbroken = x.state == State.ACTIVE && x.terms.temporal == Temporal.MAINTAIN && x.terms.deadline == NEVER
        Patch("release", mapOf(c to x.close(if (unbroken) State.DISCHARGED else State.RELEASED, now)), commitment = c, exempt = setOf(c))
    }

    fun cancel(c: Id, notice: String, now: Long) = submit(now) { view ->
        val x = view.get<Commits>(c)
        if (x.state !in OPEN) refuse(c, "$c is not open")
        val withdrawing = view.actsFor(actor, x.debtor) || (x.debtor == null && view.actsFor(actor, x.creditor))
        if (!withdrawing) refuse(c, "only the debtor cancels")
        Patch("cancel", mapOf(c to x.close(State.CANCELLED, now, notice)), commitment = c, exempt = setOf(c))
    }

    fun escalate(c: Id, to: Id, witness: String, now: Long) = submit(now) { view ->
        val x = view.get<Commits>(c)
        if (x.state != State.VIOLATED && x.state !in LIVE) refuse(c, "$c is ${x.state}")
        val party = actor == Store.ENGINE || view.actsFor(actor, x.debtor) || view.actsFor(actor, x.creditor)
        if (!party) refuse(c, "only a party escalates")
        Patch("escalate", mapOf(c to x.escalate(witness, now, setOf(to))), commitment = c)
    }

    fun reserve(c: Id, resource: Id, amount: Long, from: Long, until: Long, now: Long) = creating(now) { id, view ->
        val x = view.get<Commits>(c)
        if (x.state !in OPEN) refuse(c, "$c is closed")
        if (!view.actsFor(actor, x.debtor) && !view.actsFor(actor, x.creditor)) refuse(c, "only a party reserves for $c")
        Patch("reserve", mapOf(id to Uses(c, resource, amount, from, until, grant, x.scope)), commitment = c)
    }
}
```

workgraph/Liveness.kt | **Listing 16.** Heartbeats, timers and the reactor. Kotlin 2.4

```
package workgraph

import kotlin.concurrent.atomics.AtomicLong
import kotlin.concurrent.atomics.AtomicReference

class Liveness {
    private class Slot(since: Long) {
        val reported = AtomicLong(since)
    }

    private val slots = AtomicReference(emptyMap<Id, Slot>())

    fun track(c: Id, now: Long) = slots.update { if (c in it) it else it + (c to Slot(now)) }

    fun untrack(c: Id) = slots.update { it - c }

    fun renew(c: Id, now: Long) {
        slots.load()[c]?.reported?.store(now)
    }

    fun lastReport(c: Id): Long? = slots.load()[c]?.reported?.load()

    fun tracked(): Set<Id> = slots.load().keys
}

class Rule(val verb: String, val applies: (View, Commits) -> Boolean, val change: (Commits, Long) -> Commits)

class Reactor(private val graph: WorkGraph) : Listener {
    private val reactions = listOf(
        Rule("activate", { view, x -> x.state == State.CONDITIONAL && x.terms.antecedent?.let(view::holds) == true }) { x, _ ->
            x.copy(state = State.ACTIVE)
        },
        Rule("discharge", { view, x ->
```

```
            x.state == State.ACTIVE && x.terms.temporal == Temporal.ACHIEVE && !x.terms.creditorDecides &&
                view.now <= x.terms.deadline && view.holds(x.terms.acceptanceQuery)
        }) { x, now -> x.close(State.DISCHARGED, now) },
        Rule("violate", { view, x ->
            x.state == State.ACTIVE && x.terms.temporal == Temporal.MAINTAIN && view.truth(x.terms.acceptanceQuery) ==
                Truth.FAILS
        }) { x, now -> x.close(State.VIOLATED, now, "the invariant no longer holds").escalate("invariant observed false", now,
            setOf(x.creditor)) },
    )

    private val lapsed = Rule("lapse", { _, x -> x.state == State.ACTIVE && !x.atRisk }) { x, _ -> x.copy(atRisk = true) }

    private val timers = listOf(
        Rule("deem", { view, x -> x.inReview && x.terms.deemedAccepted && view.now > (x.reviewSince ?: NEVER) + x.terms.review
            }) { x, now ->
            x.close(State.DISCHARGED, now, "deemed accepted")
        },
        Rule("review-lapse", { view, x -> x.inReview && x.escalation == null && view.now > (x.reviewSince ?: NEVER) +
            x.terms.review }) { x, now ->
            x.escalate("review window lapsed", now, setOf(x.terms.arbiter ?: x.creditor))
        },
        Rule("miss", { view, x ->
            x.state == State.ACTIVE && !x.inReview && x.terms.temporal == Temporal.ACHIEVE && view.now > x.terms.deadline
        }) { x, now -> x.close(State.VIOLATED, now, "deadline passed").escalate("deadline passed", now, setOf(x.creditor)) },
        Rule("horizon", { view, x ->
            x.state == State.ACTIVE && x.terms.temporal == Temporal.MAINTAIN && x.terms.deadline != NEVER &&
                view.now >= x.terms.deadline && view.holds(x.terms.acceptanceQuery)
        }) { x, now -> x.close(State.DISCHARGED, now) },
    )

    override fun committed(snapshot: Snapshot, patch: Patch, events: List<Event>, now: Long) {
        for ((id, element) in patch.writes) {
            if (element is Commits) if (element.state == State.ACTIVE) graph.liveness.track(id, now) else
                graph.liveness.untrack(id)
            if (element is Grants && element.revoked) revoked(snapshot, id, element, now)
        }
        val touched = patch.footprint.flatMapTo(HashSet()) { snapshot.guardsOf(it) } + patch.writes.keys.filter { snapshot[it]
            is Commits }
        for (c in touched) {
            val x = snapshot[c] as? Commits ?: continue
            reactions.firstOrNull { it.applies(View(snapshot, now), x) }?.let { fire(c, it, now) }
        }
    }

    fun tick(now: Long) {
        val snapshot = graph.store.snapshot()
        for (c in graph.liveness.tracked()) {
            val x = snapshot[c] as? Commits ?: continue
            val quiet = now - (graph.liveness.lastReport(c) ?: now)
            val rule = timers.firstOrNull { it.applies(View(snapshot, now), x) }
                ?: lapsed.takeIf { x.terms.heartbeat != NEVER && quiet > x.terms.heartbeat && it.applies(View(snapshot, now),
                    x) }
            rule?.let { fire(c, it, now) }
        }
    }

    private fun revoked(snapshot: Snapshot, grant: Id, granted: Grants, now: Long) {
        val rule = Rule("revoked", { _, x -> x.authority == grant && x.state in LIVE && !x.atRisk }) { x, at ->
            x.copy(atRisk = true).escalate("grant $grant was revoked", at, setOf(granted.grantor, x.creditor))
        }
        for ((c, x) in snapshot.all<Commits>()) if (rule.applies(View(snapshot, now), x)) fire(c, rule, now)
    }

    private fun fire(c: Id, rule: Rule, now: Long) = graph.engine.submit(now) { view ->
        val x = view.get<Commits>(c)
        if (!rule.applies(view, x)) refuse(c, "${rule.verb} no longer applies to $c")
        Patch(rule.verb, mapOf(c to rule.change(x, now)), commitment = c, exempt = setOf(c))
    }
}
```

workgraph/Queries.kt | **Listing 17.** Status queries, attention, Percent Plan Complete, capability and couplings. Kotlin 2.4

```
package workgraph

data class OutcomeStatus(val proposition: Id, val statement: String, val weight: Double, val states: Map<State, Int>)

data class Obligation(
    val commitment: Id,
    val debtor: Id?,
    val creditor: Id,
    val consequent: Id,
    val temporal: Temporal,
    val deadline: Long,
    val state: State,
```

```
    val inReview: Boolean,
    val atRisk: Boolean,
)

data class Attention(val item: Id, val decider: Id, val why: String, val rank: Double, val dependents: Set<Id>)

sealed interface Coupling {
    data class Dependency(val first: Id, val then: Id) : Coupling
    data class CoDesign(val one: Id, val other: Id) : Coupling
}

fun Snapshot.adopted(proposition: Id): Boolean {
    val superseded = all<Decides>().mapNotNullTo(HashSet()) { it.second.supersedes }
    val votes = all<Decides>()
        .filter { (id, decision) -> decision.choice == proposition && id !in superseded }
        .map { (_, decision) -> decision to ((get(decision.authority) as? Grants)?.rule ?: DecisionRule.Authority) }
        .toList()
    if (votes.any { (decision, rule) -> decision.stance == Stance.VETO && rule is DecisionRule.Veto && decision.decider in
        rule.holders }) return false
    return votes.filter { it.first.stance == Stance.ADOPT }.groupBy({ it.second }, { it.first.decider }).any { (rule, deciders)
        ->
        rule !is DecisionRule.Quorum || deciders.filter { it in rule.members }.toSet().size >= rule.needed
    }
}

private fun Snapshot.parts(): Map<Id, List<Id>> = all<Contributes>().map { it.second }.groupBy({ it.whole }, { it.part })

private fun closure(root: Id, next: (Id) -> Iterable<Id>): Set<Id> {
    val seen = linkedSetOf(root)
    val stack = ArrayDeque(listOf(root))
    while (stack.isNotEmpty()) for (item in next(stack.removeLast())) if (seen.add(item)) stack.addLast(item)
    return seen
}

fun Snapshot.outcomes(): List<OutcomeStatus> {
    val parts = parts()
    val chosen = all<Decides>().filter { it.second.stance == Stance.ADOPT }.mapTo(LinkedHashSet()) { it.second.choice }
    return chosen.filter { get(it) is Proposition && adopted(it) }.map { root ->
        val covered = closure(root) { parts[it].orEmpty() }
        val states = all<Commits>().filter { it.second.terms.consequent in covered }.groupingBy { it.second.state }.eachCount()
        val proposition = get(root) as Proposition
        OutcomeStatus(root, proposition.statement, proposition.weight, states)
    }
}

fun Snapshot.obligations(): List<Obligation> = all<Commits>().filter { it.second.state in LIVE }.map { (id, c) ->
    Obligation(id, c.debtor, c.creditor, c.terms.consequent, c.terms.temporal, c.terms.deadline, c.state, c.inReview, c.atRisk)
}.toList()

fun Snapshot.changes(): List<Transition> =
    transitions().filter { it.verb in setOf("amend", "release", "cancel", "decide", "violate", "miss", "escalate") }.toList()

private fun Snapshot.blockedBy(c: Id): Set<Id> =
    guardsOf(c).filterTo(HashSet()) { (get(it) as? Commits)?.state == State.CONDITIONAL && c in observedBy(it) }

private fun Snapshot.rootBlocker(c: Id): Id {
    var current = c
    val seen = HashSet<Id>()
    while (seen.add(current)) {
        val x = get(current) as? Commits ?: return current
        if (x.state != State.CONDITIONAL) return current
        current = observedBy(current).firstOrNull { (get(it) as? Commits)?.state.let { s -> s != null && s != State.DISCHARGED }
            } ?: return current
    }
    return current
}

fun Snapshot.attention(now: Long, minimumSlack: Long = 1): List<Attention> {
    val parts = parts()
    val outcomes = outcomes()
    val alerts = all<Commits>().mapNotNull { (id, c) ->
        val reviewEnds = (c.reviewSince ?: now) + c.terms.review
        when {
            c.state == State.PROPOSED && !c.consented -> Triple(id, c.creditor, "awaits consent") to c.terms.deadline
            c.inReview -> Triple(id, c.terms.arbiter.takeIf { now > reviewEnds } ?: c.creditor, "awaits acceptance") to
                reviewEnds
            c.escalation != null -> Triple(id, c.escalation.to.first(), c.escalation.witness) to c.terms.deadline
            c.state == State.VIOLATED -> Triple(id, c.creditor, "violated") to now
            c.state == State.CONDITIONAL && now > c.terms.deadline -> Triple(id, c.creditor, "overdue while blocked") to
                c.terms.deadline
            c.atRisk -> Triple(id, c.debtor ?: c.creditor, "at risk") to c.terms.deadline
            else -> null
        }
    }.toList()
    val effective = alerts.associate { (alert, due) -> alert.first to due }
    return alerts.groupBy { rootBlocker(it.first.first) }.map { (root, group) ->
        val (_, decider, why) = group.firstOrNull { it.first.first == root }?.first ?: group.first().first
```

```
        val dependents = closure(root) { blockedBy(it) } - root
        val consequent = (get(root) as? Commits)?.terms?.consequent
        val weight = outcomes.sumOf { outcome ->
            val covered = closure(outcome.proposition) { parts[it].orEmpty() }
            if (consequent == null || consequent !in covered) {
                0.0
            } else {
                outcome.weight * (1 + dependents.count { d -> (get(d) as? Commits)?.terms?.consequent?.let { it in covered } == true })
            }
        }
        val slack = maxOf((effective[root] ?: now).let { if (it == NEVER) NEVER else it - now }, minimumSlack)
        Attention(root, decider, why, weight.coerceAtLeast(1.0 + dependents.size) / slack, dependents)
    }.sortedByDescending { it.rank }
}

fun Snapshot.percentPlanComplete(from: Long, to: Long, group: (Commits) -> String = { it.debtor.toString() }): Map<String, Double> =
    all<Commits>().map { it.second }
        .filter { it.terms.temporal == Temporal.ACHIEVE && it.terms.deadline in from until to && it.state !in setOf(State.PROPOSED, State.DECLINED) }
        .groupBy(group)
        .mapValues { (_, due) -> due.count { it.state == State.DISCHARGED && (it.closedAt ?: NEVER) <= it.terms.deadline }.toDouble() / due.size }

fun Snapshot.capability(actor: Id, trustRoots: Set<Id>, depth: Int = 2): Double {
    val trusted = HashSet(trustRoots)
    var frontier: Set<Id> = trustRoots
    repeat(depth) {
        frontier = all<Grants>().filter { it.second.grantor in frontier && it.second.grantor != actor && !it.second.revoked }.mapNotNullTo(HashSet()) { (_, grant) ->
            grant.grantee.takeIf { trusted.add(it) }
        }
    }
    val judged = all<Commits>().map { it.second }.filter {
        it.debtor == actor && it.creditor != actor && it.creditor in trusted && it.state in setOf(State.DISCHARGED, State.VIOLATED)
    }.toList()
    return (judged.count { it.state == State.DISCHARGED } + 1.0) / (judged.size + 2.0)
}

fun Snapshot.couplings(): List<Coupling> {
    val found = LinkedHashSet<Coupling>()
    for ((a, x) in all<Commits>().filter { it.second.state in LIVE }) {
        for (predicate in x.terms.lease) for (b in guardsOf(predicate)) {
            if (b == a || predicate !in observedBy(b)) continue
            val y = get(b) as? Commits ?: continue
            when {
                y.terms.lease.any { it in observedBy(a) } -> found += if (a.slot < b.slot) Coupling.CoDesign(a, b) else Coupling.CoDesign(b, a)
                b !in observedBy(a) -> found += Coupling.Dependency(a, b)
            }
        }
    }
    return found.toList()
}

fun Snapshot.stale(proposition: Id): Boolean = all<Attests>().any { (_, attestation) ->
    attestation.proposition == proposition && attestation.verdict == Truth.HOLDS && attestation.about.any { lastWrite(it) > attestation.atRevision }
}
```

workgraph/Markets.kt | **Listing 18.** Modes, matching, escrow-backed contracts and the federation host log. Kotlin 2.4

```
package workgraph

const val MINUTE = 60_000L
const val HOUR = 60 * MINUTE
const val DAY = 24 * HOUR
const val WEEK = 7 * DAY

enum class Trigger { ANTECEDENT, EVIDENCE, SESSION, OFFER, DECISION, REGRESSION }

enum class RegionPolicy { EXCLUSIVE, SHARED, GUARD_ONLY }

enum class DecisionKind { AUTHORITY, QUORUM, VETO }

data class Mode(val name: String, val trigger: Trigger, val region: RegionPolicy, val decision: DecisionKind, val cadence: Long) {
    fun shape(terms: Terms): Terms = when (region) {
        RegionPolicy.EXCLUSIVE -> terms.copy(merge = MergeLaw.EXCLUSIVE)
        RegionPolicy.SHARED -> terms.copy(merge = MergeLaw.COMMUTATIVE)
        RegionPolicy.GUARD_ONLY -> terms.copy(lease = emptySet())
    }.copy(heartbeat = cadence)
```

```
    fun rule(deciders: Set<Id>): DecisionRule = when (decision) {
        DecisionKind.AUTHORITY -> DecisionRule.Authority
        DecisionKind.QUORUM -> DecisionRule.Quorum(deciders, deciders.size / 2 + 1)
        DecisionKind.VETO -> DecisionRule.Veto(deciders)
    }
}

val Modes = listOf(
    Mode("Run a playbook", Trigger.ANTECEDENT, RegionPolicy.EXCLUSIVE, DecisionKind.AUTHORITY, DAY),
    Mode("Investigate", Trigger.EVIDENCE, RegionPolicy.GUARD_ONLY, DecisionKind.AUTHORITY, DAY),
    Mode("Work together live", Trigger.SESSION, RegionPolicy.SHARED, DecisionKind.AUTHORITY, 15 * MINUTE),
    Mode("Negotiate", Trigger.OFFER, RegionPolicy.EXCLUSIVE, DecisionKind.AUTHORITY, WEEK),
    Mode("Govern", Trigger.DECISION, RegionPolicy.GUARD_ONLY, DecisionKind.QUORUM, WEEK),
    Mode("Maintain", Trigger.REGRESSION, RegionPolicy.GUARD_ONLY, DecisionKind.AUTHORITY, DAY),
)

fun Party.post(terms: Terms, now: Long) = propose(terms.copy(debtor = null, creditor = actor), now)

fun Party.offer(call: Id, terms: Terms, now: Long) = propose(terms.copy(debtor = actor), now, respondsTo = call)

fun Party.award(offer: Id, considered: List<Id>, now: Long) = decide(offer, Stance.ADOPT, now, considered)

class Contract(val work: Id, val payment: Id, val delivered: Id, val paid: Id, val escrow: Id)

fun Party.contract(worker: Id, work: Terms, price: Long, money: Id, payBy: Long, now: Long): Pair<Contract, Outcome> {
    val ids = List(5) { graph.store.allocate() }
    val contract = Contract(ids[0], ids[1], ids[2], ids[3], ids[4])
    val outcome = submit(now, key = contract.work) { view ->
        val scope = view.get<Grants>(grant).over
        val delivered = Proposition("${contract.work} is discharged", scope) { v ->
            if (v.get<Commits>(contract.work).state == State.DISCHARGED) Truth.HOLDS else Truth.FAILS
        }
        val payment = Terms(
            debtor = actor,
            creditor = worker,
            consequent = contract.paid,
            antecedent = contract.delivered,
            deadline = payBy,
            creditorDecides = true,
            arbiter = work.arbiter,
        )
        Patch(
            "contract",
            mapOf(
                contract.work to Commits(work.copy(debtor = worker, creditor = actor), grant, actor, scope),
                contract.delivered to delivered,
                contract.paid to Proposition("$price paid to $worker", scope) { Truth.HOLDS },
                contract.payment to Commits(payment, grant, actor, scope),
                contract.escrow to Uses(contract.payment, money, price, now, payBy, grant, scope),
            ),
            commitment = contract.work,
        )
    }
    return contract to outcome
}

data class Entry(val sequence: Int, val revision: Long, val verb: String, val actor: Id, val digest: String)

private fun chain(previous: String, sequence: Int, revision: Long, verb: String, actor: Id) =
    fnv("$previous|$sequence|$revision|$verb|$actor")

fun Snapshot.hostLog(c: Id): List<Entry> {
    var previous = ""
    return transitions().filter { it.commitment == c }.mapIndexed { sequence, t ->
        Entry(sequence, t.revision, t.verb, t.actor, chain(previous, sequence, t.revision, t.verb, t.actor).also { previous =
            it })
    }.toList()
}

fun verify(log: List<Entry>): Boolean {
    var previous = ""
    return log.withIndex().all { (index, entry) ->
        entry.sequence == index && entry.digest == chain(previous, index, entry.revision, entry.verb, entry.actor).also {
            previous = it }
    }
}

fun fnv(text: String): String {
    var hash = -0x340d631b7bdddcdbL
    for (byte in text.encodeToByteArray()) hash = (hash xor (byte.toLong() and 0xff)) * 0x100000001b3L
    return hash.toULong().toString(16).padStart(16, '0')
}
```

workgraph/Subjects.kt | **Listing 19.** Subject spaces, preconditions and shadow migration. Kotlin 2.4

```kotlin
package workgraph

class SubjectSpace(val name: String, val vocabulary: Vocabulary, val checks: List<Check> = emptyList())

fun Party.precondition(proposition: Id, now: Long): Pair<Id, Outcome> {
    val (id, proposed) = propose(Terms(debtor = actor, creditor = actor, consequent = proposition, temporal =
        Temporal.MAINTAIN), now)
    return id to if (proposed is Outcome.Admitted) commit(id, now) else proposed
}

enum class Disagreement { FALSE_REFUSAL, MISSED_CONFLICT }

class Migrated(val store: Store, val exposed: List<Coupling>)

class Migration(private val resolution: Map<Id, Set<Id>>) {
    val disagreements = mutableListOf<Pair<Disagreement, Witness>>()

    fun resolve(region: Set<Id>): Set<Id> = region.flatMapTo(LinkedHashSet()) { resolution[it] ?: setOf(it) }

    fun migrate(from: Snapshot, finer: Map<Id, Element>, checks: List<Check>, now: Long): Migrated {
        val carried = from.elements().associate { (id, element) ->
            id to when (element) {
                is Commits -> element.copy(terms = element.terms.copy(lease = resolve(element.terms.lease)))
                is Attests -> element.copy(about = resolve(element.about))
                else -> element
            }
        }
        val moved = Snapshot.EMPTY.next(Patch("migrate", carried + finer), from.revision + 1, Store.ENGINE, now)
        val live = moved.all<Commits>().filter { it.second.state in LIVE }.toMap()
        val observed = live.mapValues { (_, c) ->
            listOfNotNull(c.terms.acceptanceQuery, c.terms.antecedent).flatMapTo(HashSet()) { moved.evaluate(it,
                now).reads.toSet() }
        }
        val indexed = moved.next(Patch("recompute regions", live), moved.revision + 1, Store.ENGINE, now,
            observed).inherit(from)
        val before = from.couplings().toSet()
        return Migrated(Store(checks, initial = indexed), indexed.couplings().filter { it !in before })
    }

    fun shadow(old: Store, new: Store, coarse: Submission, fine: Submission): Outcome {
        val authoritative = old.submit(coarse)
        val trial = new.trial(fine)
        when {
            authoritative is Outcome.Refused && trial is Outcome.Admitted -> disagreements += Disagreement.FALSE_REFUSAL to
                authoritative.witness
            authoritative is Outcome.Admitted && trial is Outcome.Refused -> disagreements += Disagreement.MISSED_CONFLICT to
                trial.witness
            authoritative is Outcome.Admitted -> new.submit(fine)
        }
        return authoritative
    }
}
```

E.2 Tests

The tests drive the engine through the scenarios of the paper. Table 11 lists them by class; the fixture and the classes follow.

Class	Tests	What the tests establish
`AdmissionTest`	14	In the ceiling void of Figure 3, a duct moved into the tray passes the lease and fails the guard with a full witness, and a clear move is admitted. Writes into another debtor's lease are refused unless the creditor permits them; malformed geometry fails validity; a stale read and a promise formed after the base both fail freshness; expired, revoked and uncredentialled grants and grants outside their scope fail authority. An expensive guard either waits for asynchronous verification or admits the change and flags the commitment at risk. Merge laws let others add to a shared region; a standing invariant refuses breaking edits and is violated by observations; a retried submission is admitted once.
`CommitmentTest`	13	Requests and offers need the other party's consent; only the debtor commits; a conditional commitment activates when its antecedent holds; an achievement discharges when its acceptance query holds; a report records evidence; creditor-owned acceptance keeps the debtor from grading its own work; a rejection cites defects and gives back the review time; decline, cancel and release close commitments; amendments and novation need every bound party; a delegate works inside the parent's lease; an escalation names its target and witness.
`LivenessTest`	7	A lapsed heartbeat flags the commitment and a report clears it; a missed deadline violates and escalates; review pauses the debtor's clock and a lapsed review goes to the arbiter; deemed acceptance discharges; a standing commitment discharges at its horizon; revoking a grant puts its commitments at risk; a watch wakes only for its predicates.
`StoreTest`	6	Fifty agents on fifty threads reserve from one budget at once with no refusal; capacity, budgets and time intervals bound reservations; disjoint writers never conflict; a contended read-modify-write loses no update; independent changes commute; pinned readers keep their snapshot.
`QueryTest`	8	Outcomes roll up the commitments beneath them; quorum and veto rules decide adoption, and only the decider withdraws a vote; the late slab surfaces once with its twelve blocked trades; Percent Plan Complete counts promises kept on time; a self-dealt history earns no capability; couplings separate co-design from ordered dependency; an attestation goes stale when what it covered changes.
`MarketTest`	4	An open call takes offers and the award declines the rest; a contract escrows payment and pays on accepted delivery; modes shape terms and decision rules; the federation host log verifies and exposes tampering.
`MigrationTest`	2	Shadow evaluation reports a false refusal and a missed conflict; migration exposes couplings the coarse model hid and keeps identities, leases, idempotency keys and history.
`ModelTest`	5	The model has 8 kinds, 15 families and 4 modalities; every element maps to its kind or family; vocabularies start at Intent; columns are persistent; read sets hold sparse identifiers.

Table 11: The 59 tests of the Kotlin engine.

workgraph/Fixture.kt | **Listing 20.** The test fixture: a project with a scope, actors under grants, propositions and elements. Kotlin 2.4

```kotlin
package workgraph

data class Box(val x0: Int, val y0: Int, val x1: Int, val y1: Int) {
    fun overlaps(other: Box) = x0 < other.x1 && other.x0 < x1 && y0 < other.y1 && other.y0 < y1
    fun moved(dx: Int) = copy(x0 = x0 + dx, x1 = x1 + dx)
}

class Project(checks: List<Check> = emptyList(), budget: Int = 10_000) {
    val graph = WorkGraph(Store(checks, budget))
    val engine = graph.engine
    val scope: Id = engine.add(Scope("project", ROOT), 0).admitted()
    val snapshot: Snapshot get() = graph.store.snapshot()

    fun actor(
        name: String,
        role: ActorRole = ActorRole.PERSON,
        rights: Set<Right> = Right.entries.toSet(),
        budget: Map<Id, Long> = emptyMap(),
        condition: Id? = null,
        expiry: Long = NEVER,
    ): Party {
        val id = engine.add(Actor(name, role, scope), 0).admitted()
        val grant = engine.grant(id, rights, scope, 0, condition = condition, budget = budget, expiry = expiry).admitted()
        return graph.party(id, grant)
    }

    fun proposition(statement: String, weight: Double = 1.0, query: Query): Id =
        engine.add(Proposition(statement, scope, weight, query), 0).admitted()

    fun element(name: String, extent: Any?, vocabulary: Vocabulary = Construction, level: String = "Element"): Id =
        engine.add(Subject(Kind.STATE, name, vocabulary, level, extent, scope), 0).admitted()

    fun flag(name: String, value: Boolean = false): Pair<Id, Id> {
        val element = element(name, value, Files, "Subject")
        return element to proposition("$name is set") { view -> if (view.get<Subject>(element).extent == true) Truth.HOLDS else
            Truth.FAILS }
    }

    fun state(c: Id): State = (snapshot[c] as Commits).state

    fun commitment(c: Id): Commits = snapshot[c] as Commits
}

fun Pair<Id, Outcome>.admitted(): Id {
```

```
    check(second is Outcome.Admitted) { "refused: $second" }
    return first
}

fun Outcome.admitted() = check(this is Outcome.Admitted) { "refused: $this" }

fun Outcome.refusal(): Witness = (this as? Outcome.Refused)?.witness ?: error("admitted: $this")

fun Pair<Id, Outcome>.refusal(): Witness = second.refusal()

fun Party.agree(terms: Terms, debtor: Party, now: Long = 1): Id {
    val c = propose(terms, now).admitted()
    debtor.commit(c, now).admitted()
    return c
}
```

workgraph/AdmissionTest.kt | **Listing 21.** Admission in the ceiling void. Kotlin 2.4

```
package workgraph

import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.assertTrue

class AdmissionTest {
    private val boxes = mutableListOf<Id>()
    private val wellFormed = Check("boxes are well formed") { view ->
        if (boxes.all { (view.get<Subject>(it).extent as Box).let { b -> b.x0 < b.x1 && b.y0 < b.y1 } }) Truth.HOLDS else Truth.FAILS
    }

    private fun void(budget: Int = 10_000) = Void(Project(listOf(wellFormed), budget), boxes)

    class Void(val project: Project, boxes: MutableList<Id>) {
        val general = project.actor("general contractor")
        val electrical = project.actor("electrical")
        val mechanical = project.actor("mechanical")
        val tray = project.element("tray T-4", Box(0, 0, 10, 2)).also(boxes::add)
        val duct = project.element("duct D-12", Box(0, 5, 10, 7)).also(boxes::add)
        val beam = project.element("beam B-7", Box(0, 9, 10, 11)).also(boxes::add)
        val clashFree = project.proposition("T-4 is free of clashes") { view ->
            val t = view.get<Subject>(tray).extent as Box
            if (listOf(duct, beam).none { (view.get<Subject>(it).extent as Box).overlaps(t) }) Truth.HOLDS else Truth.FAILS
        }

        fun trayCommitment(timeout: Timeout = Timeout.FAIL_CLOSED) = general.agree(
            Terms(debtor = electrical.actor, creditor = general.actor, consequent = clashFree, lease = setOf(tray), deadline = 1_000, creditorDecides = true, timeout = timeout),
            electrical,
        )

        fun move(party: Party, element: Id, to: Box, now: Long = 10) = party.write(element, now) { (it as Subject).copy(extent = to) }
    }

    @Test
    fun aClashingDuctPassesTheLeaseAndFailsTheGuardWithAWitness() {
        val v = void()
        val c = v.trayCommitment()
        val witness = v.move(v.mechanical, v.duct, Box(0, 1, 10, 3)).refusal()
        assertEquals(Witness(Clause.GUARD, v.duct, c, v.electrical.actor, v.general.actor, witness.reason), witness)
    }

    @Test
    fun aClearMoveOfTheDuctIsAdmitted() {
        val v = void()
        v.trayCommitment()
        v.move(v.mechanical, v.duct, Box(0, 5, 10, 7).moved(5)).admitted()
    }

    @Test
    fun writingIntoAnotherDebtorsLeaseIsRefused() {
        val v = void()
        val c = v.trayCommitment()
        val witness = v.move(v.mechanical, v.tray, Box(0, 0, 9, 2)).refusal()
        assertEquals(Clause.LEASE, witness.clause)
        assertEquals(c, witness.commitment)
        assertEquals(v.electrical.actor, witness.holder)
        v.move(v.electrical, v.tray, Box(0, 0, 9, 2)).admitted()
    }

    @Test
    fun theCreditorCanPermitAWriteIntoTheLease() {
        val v = void()
```

```
        val c = v.trayCommitment()
        val permission = v.general.decide(c, Stance.PERMIT, 5, party = v.mechanical.actor).admitted()
        v.mechanical.write(v.tray, 10, setOf(permission)) { (it as Subject).copy(extent = Box(0, 0, 9, 2)) }.admitted()
    }

    @Test
    fun malformedGeometryFailsValidity() {
        val v = void()
        assertEquals(Clause.VALIDITY, v.move(v.mechanical, v.duct, Box(5, 5, 1, 7)).refusal().clause)
    }

    @Test
    fun aChangeBuiltOnAStaleReadIsRefused() {
        val v = void()
        val old = v.project.snapshot
        v.move(v.electrical, v.duct, Box(0, 5, 10, 8)).admitted()
        val stale = v.project.graph.store.submit(
            Submission(v.mechanical.actor, v.mechanical.grant, 11, { view ->
                val duct = view.get<Subject>(v.duct)
                Patch("write", mapOf(v.duct to duct.copy(extent = (duct.extent as Box).moved(1))))
            }, base = old),
        )
        assertEquals(Witness(Clause.FRESHNESS, v.duct, reason = stale.refusal().reason), stale.refusal())
    }

    @Test
    fun aPromiseFormedAfterTheBaseRevisionStopsAnInFlightEdit() {
        val v = void()
        val old = v.project.snapshot
        val c = v.general.agree(
            Terms(debtor = v.electrical.actor, creditor = v.general.actor, consequent = v.clashFree, lease = setOf(v.beam),
                creditorDecides = true),
            v.electrical,
        )
        val late = v.project.graph.store.submit(
            Submission(v.mechanical.actor, v.mechanical.grant, 11, {
                Patch("write", mapOf(v.beam to Subject(Kind.STATE, "beam B-7", Construction, "Element", Box(0, 9, 10, 12),
                    v.project.scope)))
            }, base = old),
        )
        assertEquals(Clause.FRESHNESS, late.refusal().clause)
        assertEquals(c, late.refusal().commitment)
    }

    @Test
    fun expiredRevokedAndUncredentialledGrantsAreRefused() {
        val project = Project()
        val wall = project.element("wall", Box(0, 0, 1, 1))
        val temp = project.actor("temp", expiry = 50)
        assertEquals(Clause.AUTHORITY, temp.write(wall, 60) { it }.refusal().clause)
        val revoked = project.actor("revoked")
        project.engine.revoke(revoked.grant, 5).admitted()
        assertEquals(Clause.AUTHORITY, revoked.write(wall, 6) { it }.refusal().clause)
        val (licence, licensed) = project.flag("engineer holds a licence")
        val engineer = project.actor("engineer", condition = licensed)
        assertEquals(licensed, engineer.write(wall, 7) { it }.refusal().predicate)
        project.engine.write(licence, 8) { (it as Subject).copy(extent = true) }.admitted()
        engineer.write(wall, 9) { it }.admitted()
    }

    @Test
    fun aGrantCoversOnlyItsScope() {
        val project = Project()
        val annex = project.engine.add(Scope("annex", project.scope), 0).admitted()
        val outside = project.engine.add(Subject(Kind.STATE, "door", Construction, "Element", Box(0, 0, 1, 1), ROOT),
            0).admitted()
        val crew = project.engine.add(Actor("crew", ActorRole.TEAM, annex), 0).admitted()
        val grant = project.engine.grant(crew, setOf(Right.WRITE), annex, 0).admitted()
        val inside = project.engine.add(Subject(Kind.STATE, "shelf", Construction, "Element", Box(0, 0, 1, 1), annex),
            0).admitted()
        val party = project.graph.party(crew, grant)
        party.write(inside, 1) { it }.admitted()
        assertEquals(Clause.AUTHORITY, party.write(outside, 2) { it }.refusal().clause)
        assertEquals(Clause.VALIDITY, party.grant(crew, setOf(Right.DECIDE), annex, 3).refusal().clause)
    }

    @Test
    fun anExpensiveGuardUnderFailClosedWaitsForAsynchronousVerification() {
        val v = void(budget = 2)
        val c = v.trayCommitment()
        val refused = v.move(v.mechanical, v.duct, Box(0, 5, 10, 7).moved(5)).refusal()
        assertEquals(Witness(Clause.GUARD, v.clashFree, c, v.electrical.actor, v.general.actor, refused.reason), refused)
        assertEquals(1, v.project.graph.store.pending().size)
        assertTrue(v.project.graph.store.verify().single() is Outcome.Admitted)
    }

    @Test
```

```
    fun anExpensiveGuardUnderOptimisticAtRiskIsAdmittedAndFlagged() {
        val v = void(budget = 2)
        val events = mutableListOf<Event>()
        v.project.graph.store.listen { _, _, happened, _ -> events += happened }
        val c = v.trayCommitment(Timeout.OPTIMISTIC_AT_RISK)
        v.move(v.mechanical, v.duct, Box(0, 5, 10, 7).moved(5)).admitted()
        assertTrue(v.project.commitment(c).atRisk)
        assertEquals(listOf<Event>(Event.AtRisk(c, v.electrical.actor, v.general.actor)), events)
    }

    @Test
    fun aMergeLawLetsOthersAddToASharedRegion() {
        val project = Project()
        val couple = project.actor("couple")
        val planner = project.actor("planner")
        val guest = project.actor("guest")
        val rsvps = project.element("RSVP list", setOf<String>(), Files, "Subject")
        val listed = project.proposition("the RSVP list exists") { view -> if (view.read(rsvps) != null) Truth.HOLDS else
            Truth.FAILS }
        couple.agree(
            Terms(debtor = planner.actor, creditor = couple.actor, consequent = listed, lease = setOf(rsvps), creditorDecides =
                true, merge = MergeLaw.COMMUTATIVE),
            planner,
        )
        guest.merge(rsvps, 5) { (it as Subject).copy(extent = (it.extent as Set<*>) + "guest") }.admitted()
        assertEquals(Clause.LEASE, guest.write(rsvps, 6) { (it as Subject).copy(extent = emptySet<String>())
            }.refusal().clause)
        assertEquals(setOf("guest"), (project.snapshot[rsvps] as Subject).extent)
    }

    @Test
    fun aStandingInvariantRefusesBreakingEditsAndIsViolatedByObservations() {
        val project = Project()
        val sre = project.actor("sre")
        val latency = project.element("p95 latency", 80, Software, "Machine")
        val fast = project.proposition("p95 stays below 100 ms") { view -> if ((view.get<Subject>(latency).extent as Int) <
            100) Truth.HOLDS else Truth.FAILS }
        val (c, outcome) = sre.precondition(fast, 1)
        outcome.admitted()
        assertEquals(Clause.GUARD, project.engine.write(latency, 2) { (it as Subject).copy(extent = 140) }.refusal().clause)
        project.engine.observe(latency, (project.snapshot[latency] as Subject).copy(extent = 140), 3).admitted()
        assertEquals(State.VIOLATED, project.state(c))
        assertEquals(setOf(sre.actor), project.commitment(c).escalation?.to)
    }

    @Test
    fun aRetriedSubmissionIsAdmittedOnce() {
        val project = Project()
        val store = project.graph.store
        val key = store.allocate()
        val owner = project.actor("owner")
        val submission = Submission(owner.actor, owner.grant, 1, { Patch("add", mapOf(key to Record("d", owner.actor, 1, "once",
            project.scope))) }, key)
        val first = store.submit(submission)
        val second = store.submit(submission)
        assertEquals(first, second)
        assertEquals(1, project.snapshot.transitions().count { it.verb == "add" && it.actor == owner.actor })
    }
}
```

workgraph/CommitmentTest.kt | **Listing 22.** The commitment lifecycle. Kotlin 2.4

```
package workgraph

import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.assertNull
import kotlin.test.assertTrue

class CommitmentTest {
    private val project = Project()
    private val client = project.actor("client")
    private val builder = project.actor("builder")
    private val done = project.flag("done")
    private val work = Terms(debtor = builder.actor, creditor = client.actor, consequent = done.second, deadline = 100)

    @Test
    fun aRequestFromTheCreditorBecomesActiveWhenTheDebtorCommits() {
        val c = client.agree(work, builder)
        assertEquals(State.ACTIVE, project.state(c))
        assertEquals(builder.grant, project.commitment(c).authority)
    }

    @Test
```

```
    fun anOfferNeedsTheCreditorsConsentBeforeTheDebtorCommits() {
        val offer = builder.propose(work, 1).admitted()
        assertEquals(Clause.VALIDITY, builder.commit(offer, 2).refusal().clause)
        client.decide(offer, Stance.ADOPT, 3).admitted()
        builder.commit(offer, 4).admitted()
        assertEquals(State.ACTIVE, project.state(offer))
    }

    @Test
    fun onlyTheDebtorCommits() {
        val c = client.propose(work, 1).admitted()
        assertEquals(Clause.VALIDITY, client.commit(c, 2).refusal().clause)
    }

    @Test
    fun aConditionalCommitmentActivatesWhenItsAntecedentHolds() {
        val (permit, issued) = project.flag("permit issued")
        val c = client.agree(work.copy(antecedent = issued), builder)
        assertEquals(State.CONDITIONAL, project.state(c))
        project.engine.write(permit, 5) { (it as Subject).copy(extent = true) }.admitted()
        assertEquals(State.ACTIVE, project.state(c))
    }

    @Test
    fun anAchievementDischargesWhenItsAcceptanceQueryHolds() {
        val c = client.agree(work, builder)
        project.engine.write(done.first, 5) { (it as Subject).copy(extent = true) }.admitted()
        assertEquals(State.DISCHARGED, project.state(c))
    }

    @Test
    fun aReportRecordsEvidenceAndTheWorkDischargesWhenItsQueryHolds() {
        val c = client.agree(work, builder)
        builder.report(c, "progress", 3).admitted()
        assertEquals(State.ACTIVE, project.state(c))
        assertTrue(project.snapshot.all<Attests>().any { it.second.proposition == done.second && it.second.verdict ==
            Truth.FAILS })
        project.engine.write(done.first, 4) { (it as Subject).copy(extent = true) }.admitted()
        assertEquals(State.DISCHARGED, project.state(c))
    }

    @Test
    fun creditorOwnedAcceptanceKeepsTheDebtorFromGradingItsOwnWork() {
        val c = client.agree(work.copy(creditorDecides = true, review = 10), builder)
        project.engine.write(done.first, 2) { (it as Subject).copy(extent = true) }.admitted()
        builder.report(c, "delivered", 3).admitted()
        assertTrue(project.commitment(c).inReview)
        assertEquals(State.ACTIVE, project.state(c))
        assertEquals(Clause.VALIDITY, builder.decide(c, Stance.ADOPT, 4).refusal().clause)
        client.decide(c, Stance.ADOPT, 5).admitted()
        assertEquals(State.DISCHARGED, project.state(c))
    }

    @Test
    fun aRejectionCitesDefectsAndGivesBackTheReviewTime() {
        val c = client.agree(work.copy(creditorDecides = true, review = 10), builder)
        builder.report(c, "delivered", 30).admitted()
        assertEquals(Clause.VALIDITY, client.decide(c, Stance.REJECT, 36).refusal().clause)
        val defect = client.add(Record(fnv("crack"), client.actor, 36, "crack in bay 3", project.scope), 36).admitted()
        client.decide(c, Stance.REJECT, 36, considered = listOf(defect)).admitted()
        assertEquals(106, project.commitment(c).terms.deadline)
        assertTrue(!project.commitment(c).inReview)
    }

    @Test
    fun declineCancelAndReleaseCloseCommitments() {
        val declined = client.propose(work, 1).admitted()
        builder.decline(declined, "booked that week", 2).admitted()
        assertEquals(State.DECLINED, project.state(declined))
        val cancelled = client.agree(work, builder)
        builder.cancel(cancelled, "notice given", 3).admitted()
        assertEquals(State.CANCELLED, project.state(cancelled))
        val released = client.agree(work, builder)
        client.release(released, 4).admitted()
        assertEquals(State.RELEASED, project.state(released))
        val standing = client.agree(work.copy(temporal = Temporal.MAINTAIN, deadline = NEVER, consequent = project.flag("safe",
            true).second), builder)
        client.release(standing, 5).admitted()
        assertEquals(State.DISCHARGED, project.state(standing))
    }

    @Test
    fun anAmendmentNeedsTheConsentOfEveryBoundParty() {
        val c = client.agree(work, builder)
        val later = work.copy(deadline = 200)
        client.amend(c, later, 2).admitted()
        assertEquals(100, project.commitment(c).terms.deadline)
```

```
        builder.amend(c, later, 3).admitted()
        assertEquals(200, project.commitment(c).terms.deadline)
        assertNull(project.commitment(c).pending)
    }

    @Test
    fun novationAlsoNeedsTheIncomingDebtor() {
        val c = client.agree(work, builder)
        val successor = project.actor("successor")
        val handover = work.copy(debtor = successor.actor)
        client.amend(c, handover, 2).admitted()
        builder.amend(c, handover, 3).admitted()
        assertEquals(builder.actor, project.commitment(c).debtor)
        successor.amend(c, handover, 4).admitted()
        assertEquals(successor.actor, project.commitment(c).debtor)
    }

    @Test
    fun aDelegateWorksInsideTheParentsLease() {
        val wall = project.element("wall", Box(0, 0, 1, 1))
        val parent = client.agree(work.copy(lease = setOf(wall)), builder)
        val agent = project.actor("agent", ActorRole.AGENT)
        val paintedQuery = project.flag("painted").second
        val child = builder.delegate(parent, Terms(debtor = agent.actor, creditor = builder.actor, consequent = paintedQuery,
            lease = setOf(wall)), 2).admitted()
        assertEquals(Clause.LEASE, agent.write(wall, 3) { (it as Subject).copy(extent = Box(0, 0, 2, 2)) }.refusal().clause)
        agent.commit(child, 4).admitted()
        agent.write(wall, 5) { (it as Subject).copy(extent = Box(0, 0, 2, 2)) }.admitted()
        assertTrue(project.snapshot.all<Contributes>().any { it.second.part == paintedQuery && it.second.whole == done.second
            })
        assertEquals(setOf(child), project.commitment(parent).delegates)
    }

    @Test
    fun anEscalationNamesItsTargetAndWitness() {
        val c = client.agree(work, builder)
        val arbiter = project.actor("arbiter")
        builder.escalate(c, arbiter.actor, "zoning clause is ambiguous", 5).admitted()
        assertEquals(setOf(arbiter.actor), project.commitment(c).escalation?.to)
    }
}
```

workgraph/LivenessTest.kt | **Listing 23.** Heartbeats, deadlines, review windows and watches. Kotlin 2.4

```
package workgraph

import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.assertFalse
import kotlin.test.assertNull
import kotlin.test.assertTrue

class LivenessTest {
    private val project = Project()
    private val client = project.actor("client")
    private val agent = project.actor("agent", ActorRole.AGENT)
    private val arbiter = project.actor("arbiter")
    private val done = project.flag("done")
    private val work = Terms(debtor = agent.actor, creditor = client.actor, consequent = done.second, deadline = 50, heartbeat
        = 10)
    private val reactor = project.graph.reactor

    @Test
    fun aLapsedHeartbeatFlagsTheCommitmentAndAReportClearsIt() {
        val c = client.agree(work, agent)
        reactor.tick(15)
        assertTrue(project.commitment(c).atRisk)
        agent.report(c, "halfway", 16).admitted()
        assertFalse(project.commitment(c).atRisk)
        reactor.tick(20)
        assertFalse(project.commitment(c).atRisk)
    }

    @Test
    fun aMissedDeadlineViolatesAndEscalatesToTheCreditor() {
        val c = client.agree(work.copy(heartbeat = NEVER), agent)
        reactor.tick(51)
        assertEquals(State.VIOLATED, project.state(c))
        assertEquals(setOf(client.actor), project.commitment(c).escalation?.to)
    }

    @Test
    fun reviewPausesTheDebtorsClockAndALapsedReviewGoesToTheArbiter() {
        val c = client.agree(work.copy(heartbeat = NEVER, creditorDecides = true, review = 20, arbiter = arbiter.actor), agent)
```

```
        agent.report(c, "delivered", 40).admitted()
        reactor.tick(55)
        assertEquals(State.ACTIVE, project.state(c))
        assertNull(project.commitment(c).escalation)
        reactor.tick(61)
        assertEquals(State.ACTIVE, project.state(c))
        assertEquals(setOf(arbiter.actor), project.commitment(c).escalation?.to)
    }

    @Test
    fun deemedAcceptanceDischargesAfterTheReviewWindow() {
        val c = client.agree(work.copy(heartbeat = NEVER, creditorDecides = true, review = 5, deemedAccepted = true), agent)
        agent.report(c, "delivered", 40).admitted()
        reactor.tick(46)
        assertEquals(State.DISCHARGED, project.state(c))
    }

    @Test
    fun anUnbrokenStandingCommitmentDischargesAtItsHorizon() {
        val safe = project.flag("site is safe", true).second
        val c = client.agree(Terms(debtor = agent.actor, creditor = client.actor, consequent = safe, temporal =
            Temporal.MAINTAIN, deadline = 30), agent)
        reactor.tick(29)
        assertEquals(State.ACTIVE, project.state(c))
        reactor.tick(30)
        assertEquals(State.DISCHARGED, project.state(c))
    }

    @Test
    fun revokingAGrantPutsItsCommitmentsAtRiskAndTellsGrantorAndCreditor() {
        val c = client.agree(work, agent)
        project.engine.revoke(agent.grant, 5).admitted()
        assertTrue(project.commitment(c).atRisk)
        assertEquals(setOf(Store.ENGINE, client.actor), project.commitment(c).escalation?.to)
    }

    @Test
    fun aWatchWakesOnlyWhenAWatchedPredicateChanges() {
        val woken = mutableListOf<Set<Id>>()
        val (permit, _) = project.flag("permit")
        val other = project.element("other", 0, Files, "Subject")
        project.graph.store.watch(setOf(permit)) { _, touched -> woken += touched }
        project.engine.write(other, 2) { (it as Subject).copy(extent = 1) }.admitted()
        project.engine.write(permit, 3) { (it as Subject).copy(extent = true) }.admitted()
        assertEquals(listOf(setOf(permit)), woken)
    }
}
```

workgraph/StoreTest.kt | **Listing 24.** Escrow and writers on threads. Kotlin 2.4

```
package workgraph

import kotlin.concurrent.thread
import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.assertTrue

class StoreTest {
    private fun task(project: Project, agent: Party): Id {
        val goal = project.flag("goal of ${agent.actor}").second
        return agent.propose(Terms(debtor = agent.actor, creditor = agent.actor, consequent = goal), 1).admitted()
    }

    @Test
    fun fiftyAgentsReserveFromOneBudgetAtOnceWithoutRetries() {
        val project = Project()
        val money = project.engine.add(Resource("compute budget", 1_000, project.scope), 0).admitted()
        val agents = List(50) { project.actor("agent $it", ActorRole.AGENT, budget = mapOf(money to 10)) }
        val tasks = agents.map { task(project, it) }
        val outcomes = arrayOfNulls<Outcome>(agents.size)
        agents.indices.map { i -> thread { outcomes[i] = agents[i].reserve(tasks[i], money, 1, 0, 100, 2).second } }.forEach {
            it.join() }
        assertTrue(outcomes.all { it is Outcome.Admitted })
        assertEquals(950, project.snapshot.available(money, 0, 100, 2))
    }

    @Test
    fun capacityBudgetsAndIntervalsBoundReservations() {
        val project = Project()
        val lift = project.engine.add(Resource("scissor lift", 2, project.scope), 0).admitted()
        val crew = project.actor("crew", budget = mapOf(lift to 3))
        val job = task(project, crew)
        crew.reserve(job, lift, 1, 0, 10, 2).admitted()
        crew.reserve(job, lift, 1, 0, 10, 2).admitted()
```

```
        assertEquals(Clause.AUTHORITY, crew.reserve(job, lift, 1, 5, 15, 2).refusal().clause)
        crew.reserve(job, lift, 1, 10, 20, 2).admitted()
        assertEquals(lift, crew.reserve(job, lift, 1, 30, 40, 2).refusal().predicate)
        crew.cancel(job, "weather", 3).admitted()
        assertEquals(2, project.snapshot.available(lift, 0, 10, 3))
    }

    @Test
    fun disjointWritersNeverConflict() {
        val project = Project()
        val writers = List(8) { project.actor("writer $it") }
        val cells = writers.map { project.element("cell of ${it.actor}", 0, Files, "Subject") }
        writers.indices.map { i ->
            thread { repeat(250) { writers[i].write(cells[i], 2) { e -> (e as Subject).copy(extent = (e.extent as Int) + 1)
                }.admitted() } }
        }.forEach { it.join() }
        assertTrue(cells.all { (project.snapshot[it] as Subject).extent == 250 })
    }

    @Test
    fun contendedReadModifyWriteLosesNoUpdate() {
        val project = Project()
        val counter = project.element("counter", 0, Files, "Subject")
        val writers = List(8) { project.actor("writer $it") }
        val refusals = java.util.concurrent.atomic.AtomicInteger()
        writers.map { writer ->
            thread {
                repeat(100) {
                    while (writer.write(counter, 2) { e -> (e as Subject).copy(extent = (e.extent as Int) + 1) } is
                        Outcome.Refused) refusals.incrementAndGet()
                }
            }
        }.forEach { it.join() }
        assertEquals(800, (project.snapshot[counter] as Subject).extent)
        println("contended counter: 800 increments, ${refusals.get()} freshness refusals retried")
    }

    @Test
    fun independentChangesCommuteInEitherOrder() {
        fun run(firstLeft: Boolean): Map<String, Any?> {
            val project = Project()
            val left = project.element("left", 1, Files, "Subject")
            val right = project.element("right", 2, Files, "Subject")
            val editor = project.actor("editor")
            val base = project.snapshot
            fun change(target: Id, value: Int) = Submission(editor.actor, editor.grant, 3, { view ->
                Patch("write", mapOf(target to view.get<Subject>(target).copy(extent = value)))
            }, base = base)
            val order = if (firstLeft) listOf(change(left, 10), change(right, 20)) else listOf(change(right, 20), change(left,
                10))
            order.forEach { project.graph.store.submit(it).admitted() }
            return project.snapshot.all<Subject>().associate { (_, s) -> s.name to s.extent }
        }
        assertEquals(run(firstLeft = true), run(firstLeft = false))
    }

    @Test
    fun readersKeepTheirSnapshotWhileWritersPublish() {
        val project = Project()
        val cell = project.element("cell", 0, Files, "Subject")
        val writer = project.actor("writer")
        val pinned = project.snapshot
        repeat(10) { writer.write(cell, 2) { e -> (e as Subject).copy(extent = (e.extent as Int) + 1) }.admitted() }
        assertEquals(0, (pinned[cell] as Subject).extent)
        assertEquals(10, (project.snapshot[cell] as Subject).extent)
        assertEquals(pinned.revision + 10, project.snapshot.revision)
    }
}
```

workgraph/QueryTest.kt | **Listing 25.** Status queries, attention, Percent Plan Complete and capability. Kotlin 2.4

```
package workgraph

import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.assertFalse
import kotlin.test.assertTrue

class QueryTest {
    private val project = Project()
    private val owner = project.actor("owner")
    private val crew = project.actor("crew")
    private val trivially = project.proposition("trivially true") { Truth.HOLDS }

```

```
    private fun contributes(part: Id, whole: Id) = project.engine.add(Contributes(part, whole, project.scope), 0).admitted()

    private fun set(flag: Id, now: Long) = project.engine.write(flag, now) { (it as Subject).copy(extent = true) }.admitted()

    private fun council(size: Int, rule: (Set<Id>) -> DecisionRule): List<Party> {
        val members = List(size) { project.engine.add(Actor("member $it", ActorRole.PERSON, project.scope), 0).admitted() }
        val decided = rule(members.toSet())
        return members.map { project.graph.party(it, project.engine.grant(it, setOf(Right.DECIDE), project.scope, 0, rule =
            decided).admitted()) }
    }

    @Test
    fun outcomesRollUpTheCommitmentsBeneathThem() {
        val ship = project.proposition("version 1 ships", weight = 2.0) { Truth.UNKNOWN }
        val (apiFlag, api) = project.flag("api built")
        val ui = project.flag("ui built").second
        contributes(api, ship)
        contributes(ui, ship)
        owner.decide(ship, Stance.ADOPT, 1).admitted()
        val first = owner.agree(Terms(crew.actor, owner.actor, api), crew)
        owner.agree(Terms(crew.actor, owner.actor, ui), crew)
        assertEquals(mapOf(State.ACTIVE to 2), project.snapshot.outcomes().single().states)
        set(apiFlag, 2)
        assertEquals(State.DISCHARGED, project.state(first))
        val status = project.snapshot.outcomes().single()
        assertEquals(ship, status.proposition)
        assertEquals(mapOf(State.DISCHARGED to 1, State.ACTIVE to 1), status.states)
    }

    @Test
    fun aQuorumAdoptsWithEnoughVotesAndOnlyTheDeciderWithdrawsAVote() {
        val board = council(3) { DecisionRule.Quorum(it, 2) }
        val plan = project.proposition("adopt the zoning plan") { Truth.UNKNOWN }
        board[0].decide(plan, Stance.ADOPT, 1).admitted()
        assertFalse(project.snapshot.adopted(plan))
        val second = board[1].decide(plan, Stance.ADOPT, 2).admitted()
        assertTrue(project.snapshot.adopted(plan))
        assertEquals(Clause.VALIDITY, board[2].decide(plan, Stance.REJECT, 3, supersedes = second).refusal().clause)
        board[1].decide(plan, Stance.REJECT, 3, supersedes = second).admitted()
        assertFalse(project.snapshot.adopted(plan))
    }

    @Test
    fun aVetoHolderBlocksAdoptionUntilItWithdrawsTheVeto() {
        val plan = project.proposition("demolish the annex") { Truth.UNKNOWN }
        owner.decide(plan, Stance.ADOPT, 1).admitted()
        assertTrue(project.snapshot.adopted(plan))
        val heritage = council(1) { DecisionRule.Veto(it) }.single()
        val veto = heritage.decide(plan, Stance.VETO, 2).admitted()
        assertFalse(project.snapshot.adopted(plan))
        heritage.decide(plan, Stance.PERMIT, 3, supersedes = veto).admitted()
        assertTrue(project.snapshot.adopted(plan))
    }

    @Test
    fun aLateSlabSurfacesOnceWithItsTwelveBlockedTrades() {
        val concrete = project.actor("concrete")
        val trades = project.actor("trades")
        val strength = project.flag("level 4 slab reaches design strength").second
        val slab = owner.agree(Terms(concrete.actor, owner.actor, strength, deadline = 10), concrete)
        val accepted = project.proposition("level 4 slab is accepted") { view ->
            if (view.get<Commits>(slab).state == State.DISCHARGED) Truth.HOLDS else Truth.FAILS
        }
        val blocked = List(12) { i ->
            owner.agree(Terms(trades.actor, owner.actor, project.flag("trade $i fitted out").second, antecedent = accepted,
                deadline = 20), trades)
        }
        val signage = owner.agree(Terms(crew.actor, owner.actor, project.flag("signage installed").second, deadline = 100,
            heartbeat = 5), crew)
        assertTrue(blocked.all { project.state(it) == State.CONDITIONAL })
        project.graph.reactor.tick(30)
        val queue = project.snapshot.attention(30)
        assertEquals(listOf(slab, signage), queue.map { it.item })
        assertEquals(blocked.toSet(), queue.first().dependents)
        assertEquals(owner.actor, queue.first().decider)
        assertEquals("at risk", queue.last().why)
    }

    @Test
    fun percentPlanCompleteCountsPromisesKeptOnTime() {
        val bays = List(4) { project.flag("framing bay $it") }
        val promises = bays.mapIndexed { i, (_, framed) ->
            owner.agree(Terms(crew.actor, owner.actor, framed, deadline = 10L + 2 * i, category = "framing"), crew)
        }
        bays.take(3).forEachIndexed { i, (flag, _) -> set(flag, 5L + i) }
        project.graph.reactor.tick(17)
        assertEquals(State.VIOLATED, project.state(promises[3]))
```

```
        assertEquals(mapOf("framing" to 0.75), project.snapshot.percentPlanComplete(0, 20) { it.terms.category })
    }

    @Test
    fun selfDealtHistoryEarnsNoCapability() {
        val board = project.actor("licensing board")
        val client = project.actor("client")
        board.grant(client.actor, setOf(Right.DECIDE), project.scope, 0).admitted()
        val worker = project.actor("worker")
        val cheat = project.actor("cheat")
        val impossible = project.proposition("never happens") { Truth.FAILS }
        fun owe(creditor: Party, debtor: Party, consequent: Id = trivially, deadline: Long = NEVER) =
            creditor.agree(Terms(debtor.actor, creditor.actor, consequent, deadline = deadline), debtor)
        repeat(2) { owe(client, worker) }
        owe(client, worker, impossible, deadline = 5)
        owe(client, cheat, impossible, deadline = 5)
        repeat(10) { i ->
            val fake = project.engine.add(Actor("fake client $i", ActorRole.PERSON, project.scope), 0).admitted()
            owe(project.graph.party(fake, cheat.grant(fake, setOf(Right.DECIDE), project.scope, 0).admitted()), cheat)
        }
        project.graph.reactor.tick(6)
        assertEquals(10, project.snapshot.all<Commits>().count { it.second.debtor == cheat.actor && it.second.state ==
            State.DISCHARGED })
        for (roots in listOf(setOf(board.actor), setOf(Store.ENGINE))) {
            assertEquals(0.6, project.snapshot.capability(worker.actor, roots), 1e-9)
            assertEquals(1.0 / 3, project.snapshot.capability(cheat.actor, roots), 1e-9)
        }
    }

    @Test
    fun couplingsSeparateCoDesignFromOrderedDependency() {
        val tray = project.element("tray T-4", Box(0, 0, 10, 2))
        val duct = project.element("duct D-12", Box(0, 5, 10, 7))
        val head = project.element("sprinkler head S-3", Box(20, 5, 22, 7))
        fun clear(a: Id, b: Id) = project.proposition("$a is clear of $b") { view ->
            val mine = view.get<Subject>(a).extent as Box
            val theirs = view.get<Subject>(b).extent as Box
            if (mine.overlaps(theirs)) Truth.FAILS else Truth.HOLDS
        }
        fun place(debtor: Party, element: Id, other: Id) =
            owner.agree(Terms(debtor.actor, owner.actor, clear(element, other), lease = setOf(element), creditorDecides = true),
                debtor)
        val t = place(project.actor("electrical"), tray, duct)
        val d = place(project.actor("mechanical"), duct, tray)
        val s = place(project.actor("fire protection"), head, duct)
        assertEquals(setOf(Coupling.CoDesign(t, d), Coupling.Dependency(d, s)), project.snapshot.couplings().toSet())
    }

    @Test
    fun anAttestationGoesStaleWhenWhatItCoveredChanges() {
        val spec = project.element("pump spec", "rev A", Files, "Subject")
        val written = project.proposition("the pump spec is written") { view ->
            if (view.get<Subject>(spec).extent != null) Truth.HOLDS else Truth.FAILS
        }
        val c = owner.agree(Terms(crew.actor, owner.actor, written, creditorDecides = true), crew)
        crew.report(c, "spec attached", 2, about = setOf(spec)).admitted()
        assertFalse(project.snapshot.stale(written))
        owner.write(spec, 3) { (it as Subject).copy(extent = "rev B") }.admitted()
        assertTrue(project.snapshot.stale(written))
    }
}
```

workgraph/MarketTest.kt | **Listing 26.** Open calls, contracts, modes and the host log. Kotlin 2.4

```
package workgraph

import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.assertFalse
import kotlin.test.assertTrue

class MarketTest {
    private val project = Project()
    private val client = project.actor("client")

    @Test
    fun anOpenCallTakesOffersAndTheAwardDeclinesTheRest() {
        val vendors = List(2) { project.actor("vendor $it") }
        val inspected = project.flag("roof inspected").second
        assertEquals(Clause.VALIDITY, vendors[0].propose(Terms(null, client.actor, inspected), 1).refusal().clause)
        val call = client.post(Terms(null, client.actor, inspected, deadline = 50), 1).admitted()
        assertEquals(Clause.VALIDITY, vendors[0].offer(call, Terms(null, vendors[1].actor, inspected), 2).refusal().clause)
        val offers = vendors.map { it.offer(call, Terms(null, client.actor, inspected, deadline = 40), 2).admitted() }
        assertEquals(listOf(false, false), offers.map { project.commitment(it).consented })
```

```
        client.award(offers[1], offers, 3).admitted()
        assertEquals(State.DECLINED, project.state(call))
        assertEquals(State.DECLINED, project.state(offers[0]))
        vendors[1].commit(offers[1], 4).admitted()
        assertEquals(State.ACTIVE, project.state(offers[1]))
        assertEquals(Clause.VALIDITY, vendors[0].commit(offers[0], 4).refusal().clause)
    }

    @Test
    fun aContractEscrowsPaymentAndPaysOnAcceptedDelivery() {
        val money = project.engine.add(Resource("project account", 1_000, project.scope), 0).admitted()
        val payer = project.actor("payer", budget = mapOf(money to 500))
        val worker = project.actor("worker")
        val (flag, shipped) = project.flag("feature shipped")
        val work = Terms(null, payer.actor, shipped, deadline = 50)
        assertEquals(Clause.AUTHORITY, payer.contract(worker.actor, work, 600, money, 100, 1).second.refusal().clause)
        val (contract, outcome) = payer.contract(worker.actor, work, 300, money, 100, 1)
        outcome.admitted()
        assertEquals(700, project.snapshot.available(money, 0, 100, 1))
        worker.decide(contract.payment, Stance.ADOPT, 2).admitted()
        payer.commit(contract.payment, 2).admitted()
        assertEquals(State.CONDITIONAL, project.state(contract.payment))
        worker.commit(contract.work, 3).admitted()
        worker.write(flag, 10) { (it as Subject).copy(extent = true) }.admitted()
        assertEquals(State.DISCHARGED, project.state(contract.work))
        assertEquals(State.ACTIVE, project.state(contract.payment))
        payer.report(contract.payment, "transfer of 300 sent", 11).admitted()
        worker.decide(contract.payment, Stance.ADOPT, 12).admitted()
        assertEquals(State.DISCHARGED, project.state(contract.payment))
        assertEquals(300, project.snapshot.spent(payer.grant, money))
        assertTrue(project.snapshot.liveHolds(money, 12).isEmpty())
    }

    @Test
    fun modesShapeTermsAndDecisionRules() {
        val inspected = project.flag("roof inspected").second
        val terms = Terms(null, client.actor, inspected, lease = setOf(inspected))
        assertEquals(6, Modes.size)
        val live = Modes.single { it.name == "Work together live" }.shape(terms)
        assertEquals(MergeLaw.COMMUTATIVE, live.merge)
        assertEquals(15 * MINUTE, live.heartbeat)
        assertEquals(emptySet(), Modes.single { it.name == "Investigate" }.shape(terms).lease)
        val five = List(5) { Id(100 + it) }.toSet()
        assertEquals(DecisionRule.Quorum(five, 3), Modes.single { it.name == "Govern" }.rule(five))
    }

    @Test
    fun theHostLogVerifiesAndExposesTampering() {
        val vendor = project.actor("vendor")
        val (flag, inspected) = project.flag("roof inspected")
        val c = client.agree(Terms(vendor.actor, client.actor, inspected, creditorDecides = true), vendor)
        vendor.write(flag, 2) { (it as Subject).copy(extent = true) }.admitted()
        vendor.report(c, "inspection report", 3).admitted()
        client.decide(c, Stance.ADOPT, 4).admitted()
        val log = project.snapshot.hostLog(c)
        assertEquals(listOf("propose", "commit", "report", "decide"), log.map { it.verb })
        assertTrue(verify(log))
        assertFalse(verify(log.toMutableList().also { it[1] = it[1].copy(verb = "cancel") }))
        assertFalse(verify(log.drop(1)))
    }
}
```

workgraph/MigrationTest.kt | **Listing 27.** Shadow migration. Kotlin 2.4

```
package workgraph

import kotlin.test.Test
import kotlin.test.assertEquals

class MigrationTest {
    private val project = Project()
    private val owner = project.actor("owner")
    private val store = project.graph.store

    private fun Party.writes(target: Id, now: Long, value: Any?) = Submission(actor, grant, now, { view ->
        Patch("write", mapOf(target to view.get<Subject>(target).copy(extent = value)))
    })

    private fun subject(name: String, extent: Any?) = Subject(Kind.STATE, name, Construction, "Element", extent, project.scope)

    private fun clear(a: Id, b: Id) = Query { view ->
        if ((view.get<Subject>(a).extent as Box).overlaps(view.get<Subject>(b).extent as Box)) Truth.FAILS else Truth.HOLDS
    }

```

```
    @Test
    fun shadowEvaluationReportsFalseRefusalsAndMissedConflicts() {
        val electrician = project.actor("electrician")
        val plumber = project.actor("plumber")
        val ductwork = project.actor("ductwork")
        val floor = project.element("floor 3", emptySet<String>(), level = "Spatial")
        val dry = project.proposition("floor 3 is dry") { view ->
            if ((view.get<Subject>(floor).extent as Set<*>).isEmpty()) Truth.HOLDS else Truth.FAILS
        }
        val ceiling = project.element("level 3 ceiling services", "rev A", level = "System")
        val coordinated = project.proposition("the ceiling is coordinated") { view ->
            if (view.get<Subject>(ceiling).extent != "clash") Truth.HOLDS else Truth.FAILS
        }
        electrician.precondition(dry, 1).admitted()
        electrician.precondition(coordinated, 1).admitted()

        val east = store.allocate()
        val west = store.allocate()
        val tray = store.allocate()
        val duct = store.allocate()
        val finer = mapOf(
            east to subject("floor 3 east", false),
            west to subject("floor 3 west", false),
            tray to subject("tray T-4", Box(0, 0, 10, 2)),
            duct to subject("duct D-12", Box(0, 5, 10, 7)),
            dry to Proposition("floor 3 east is dry", project.scope) { view ->
                if (view.get<Subject>(east).extent == false) Truth.HOLDS else Truth.FAILS
            },
            coordinated to Proposition("tray T-4 is clear of duct D-12", project.scope, query = clear(tray, duct)),
        )
        val migration = Migration(mapOf(floor to setOf(east, west), ceiling to setOf(tray, duct)))
        val migrated = migration.migrate(project.snapshot, finer, emptyList(), 2)

        val wet = migration.shadow(store, migrated.store, plumber.writes(floor, 3, setOf("west")), plumber.writes(west, 3,
            true))
        assertEquals(Clause.GUARD, wet.refusal().clause)
        migration.shadow(store, migrated.store, ductwork.writes(ceiling, 4, "rev B"), ductwork.writes(duct, 4, Box(0, 1, 10,
            3))).admitted()
        assertEquals(listOf(Disagreement.FALSE_REFUSAL, Disagreement.MISSED_CONFLICT), migration.disagreements.map { it.first
            })
        assertEquals(Clause.GUARD, migration.disagreements.last().second.clause)
    }

    @Test
    fun migrationExposesHiddenCouplingsAndKeepsIdsLeasesAndHistory() {
        val electrical = project.actor("electrical")
        val mechanical = project.actor("mechanical")
        val trays = project.element("cable trays", "level 3", level = "System")
        val ducts = project.element("ductwork", "level 3", level = "System")
        fun installed(subject: Id) = project.proposition("$subject installed") { view ->
            if (view.get<Subject>(subject).extent == "installed") Truth.HOLDS else Truth.FAILS
        }
        val trayPlaced = installed(trays)
        val ductPlaced = installed(ducts)
        val t = owner.agree(Terms(electrical.actor, owner.actor, trayPlaced, lease = setOf(trays), creditorDecides = true),
            electrical)
        val d = owner.agree(Terms(mechanical.actor, owner.actor, ductPlaced, lease = setOf(ducts), creditorDecides = true),
            mechanical)
        val before = project.snapshot
        assertEquals(emptyList(), before.couplings())

        val tray = store.allocate()
        val duct = store.allocate()
        val finer = mapOf(
            tray to subject("tray T-4", Box(0, 0, 10, 2)),
            duct to subject("duct D-12", Box(0, 5, 10, 7)),
            trayPlaced to Proposition("tray T-4 is placed clear of duct D-12", project.scope, query = clear(tray, duct)),
            ductPlaced to Proposition("duct D-12 is placed clear of tray T-4", project.scope, query = clear(duct, tray)),
        )
        val migrated = Migration(mapOf(trays to setOf(tray), ducts to setOf(duct))).migrate(before, finer, emptyList(), 5)
        val after = migrated.store.snapshot()
        assertEquals(listOf<Coupling>(Coupling.CoDesign(t, d)), migrated.exposed)
        assertEquals(t, after.leaseHolder(tray))
        assertEquals(d, after.leaseHolder(duct))
        assertEquals(before.admitted(t), after.admitted(t))
        assertEquals(before.transitions().count() + 2, after.transitions().count())
    }
}
```

workgraph/ModelTest.kt | **Listing 28.** The model, columns and read sets. Kotlin 2.4

```
package workgraph

import kotlin.test.Test
```

```
import kotlin.test.assertEquals
import kotlin.test.assertFailsWith
import kotlin.test.assertFalse
import kotlin.test.assertNull
import kotlin.test.assertTrue

class ModelTest {
    @Test
    fun theModelHasEightKindsFifteenFamiliesAndFourModalities() {
        assertEquals(8, Kind.entries.size)
        assertEquals(15, Family.entries.size)
        assertEquals(4, Modality.entries.size)
    }

    @Test
    fun everyElementMapsToItsKindOrFamily() {
        val x = Id(9)
        val terms = Terms(x, x, x, lease = setOf(Id(10)))
        val nodes = listOf(
            Scope("site", null),
            Actor("agent", ActorRole.AGENT, x),
            Proposition("p", x) { Truth.HOLDS },
            Record("digest", x, 0, "content", x),
            Resource("crane", 1, x),
            Subject(Kind.OPERATOR, "parse", Software, "Program", null, x),
        )
        assertEquals(listOf(Kind.SCOPE, Kind.ACTOR, Kind.PROPOSITION, Kind.RECORD, Kind.RESOURCE, Kind.OPERATOR), nodes.map {
            it.kind })
        val assertions = listOf(
            Commits(terms, x, x, x),
            Attests(x, x, x, Truth.HOLDS, emptySet(), 0, x),
            Decides(x, x, x, Stance.ADOPT, emptyList(), null, null, x),
            Grants(x, x, emptySet(), x),
            Contributes(x, x, x),
            Uses(x, x, 1, 0, 1, x, x),
            Structural(Family.REFINES, listOf(x, x), x),
        )
        assertEquals(
            listOf(Family.COMMITS, Family.ATTESTS, Family.DECIDES, Family.GRANTS, Family.CONTRIBUTES, Family.USES,
                Family.REFINES),
            assertions.map { it.family },
        )
        assertEquals(Id(10), assertions.first().ends().last())
    }

    @Test
    fun vocabulariesStartAtIntent() {
        assertEquals(listOf(7, 5, 4, 2), listOf(Software, Construction, Curriculum, Files).map { it.levels.size })
        assertEquals(6, Software.rank("Machine"))
        assertFailsWith<IllegalArgumentException> { Vocabulary("bad", listOf("Design")) }
        assertFailsWith<IllegalArgumentException> { Construction.rank("Program") }
    }

    @Test
    fun columnsArePersistentAcrossChunks() {
        val empty = Column<String>()
        val one = empty.with(mapOf(Id(3) to "three", Id(5_000) to "five thousand"))
        val two = one.with(mapOf(Id(3) to null, Id(4) to "four"))
        assertNull(empty[Id(3)])
        assertEquals("three", one[Id(3)])
        assertEquals(listOf(Id(3) to "three", Id(5_000) to "five thousand"), one.entries().toList())
        assertEquals(listOf(Id(4) to "four", Id(5_000) to "five thousand"), two.entries().toList())
    }

    @Test
    fun predicateSetsHoldSparseIds() {
        val set = PredicateSet()
        listOf(1, 64, 65, 4_000).forEach { set.add(Id(it)) }
        assertTrue(Id(65) in set)
        assertFalse(Id(2) in set)
        assertFalse(Id(100_000) in set)
        assertEquals(listOf(1, 64, 65, 4_000), set.ids().map { it.slot }.toList())
    }
}
```