Scoping and Isolation in Shared Decision Stores

kgai maintainers

kgai.dev · team@kgai.dev · August 2026

ABSTRACT

kgai stores a team's engineering decisions in an append-only log projected into a graph of domain elements. This report examines where the boundaries of such a store lie and what actually holds them. The default boundary is the repository, one store per project. Widening it is an explicit act, because a committed configuration file may request a shared store but decides nothing until a person approves it on each machine. Within a store, recall is queried per element or per path pattern, so an agent can be confined to a subgraph through its instructions. We are explicit about the nature of that confinement. It is advisory filtering, not enforcement, since anything that can read the repository or the synchronization bucket can read every decision in the store. We separate the properties kgai does enforce, concerning identity and provenance, from the confidentiality it inherits from its substrate, discuss monorepo and multi-repository considerations, and treat enforcement as a possible future direction while promising nothing.

1. Introduction

A shared memory is only as trustworthy as its boundaries. When a team records its engineering decisions into a store that agents read before changing code, three questions follow immediately. Which decisions belong in one store, who can read them, and which of the answers to the first two questions are enforced rather than merely agreed upon. The protection literature has insisted on that last distinction for decades. A convention that cooperating parties follow does not constrain a party who ignores it, and conflating the two is a classic design failure [1, 2].

This report describes the scoping and isolation model of kgai, a local-first system that records decisions into an append-only log with a derived graph projection. We describe the default boundary, the opt-in path to a wider one, the query-level scoping available inside a store, and then draw the honest line between what is advisory and what is enforced. We are the maintainers of the system, and the report's purpose is precision about limits rather than advocacy.

2. The Default Boundary: One Store per Repository

By default each project keeps its own decision log in a store directory inside the project, created automatically on first use and excluded from the project's version control. The boundary is the repository,

and it is per project rather than per branch. A decision recorded on a feature branch is visible from the main branch immediately, and worktree checkouts resolve to the main worktree's store, so switching to a worktree does not present an empty graph and a branch cannot repoint its worktree at a different store.

The repository is a defensible default because it is where a decision's subject usually lives. A repository commonly approximates a product or a bounded context in the domain-driven sense, a unit within which names and models are expected to cohere [3]. The documentation calls this the right default for open source and for teams whose repositories are genuinely separate products. The default also has a protective consequence. A repository that is never enrolled in anything wider keeps its own log, so a personal side project cannot land in a company graph by accident.

3. Widening the Boundary: Opt-In Shared Stores

The default fails a specific organizational reality. Where the same people work across several repositories, decisions cross repository lines, and the documented example is a payments responsibility moving between two services, one decision about two repositories. Splitting that decision across two logs means no agent ever sees the whole story. For this case kgai supports a shared store, in which the repositories that belong together record into one graph.

The store location is an ordinary configuration key resolved through three layers, most specific first. A session layer inside the store itself, a project layer committed to the repository, and a global per-machine layer, with an environment variable overriding all three for one-off runs. The committed project layer is the interesting one, because it is the only layer nobody on the reading machine wrote. It comes in with the clone, authored by whoever created the repository. kgai therefore treats it as untrusted until approved. The configuration reports it as pending approval, the session surfaces what the file asks for, and a person accepts or dismisses it explicitly. The approval is bound to what the file requests, the values of the store path and the capture rules, rather than to its bytes, so reformatting asks nothing new while a changed request asks again. Each approval is recorded per machine and per user in a file that never synchronizes, since consent that traveled with the clone would arrive pre-given for exactly the people the gate protects. While the question is open, kgai also refrains from creating a local store, so a later decision does not orphan one. The trust report in this series (TR-2026-08) treats this mechanism and its threat model in full.

A global setting can enroll every repository on a machine at once. The documentation is explicit that this is rarely what one wants, since it catches repositories never meant to be enrolled, and that it is defensible only on machines that exist for one purpose, such as managed development boxes or continuous integration.

Sharing with a team remote is a further, separate opt-in, and it belongs to the store rather than to any repository. The remote cannot be set in the committed file at all. One physical log cannot push to different destinations depending on which enrolled repository a session started in, and a cloned file must never

dictate where a developer's decisions travel. The remote is configured once per store, against an S3-compatible bucket the team owns.

4. Scoping Queries Within a Store

Inside a store, the scoping instrument is the query. Recall is asked per node. An agent about to modify an area requests the decisions that shaped that area, by element or by path pattern, rather than the whole history. Because recall is parameterized this way, an agent can be pinned to a subgraph by prompt. The committed configuration carries a capture-and-recall convention that reaches every clone's agent at session start, framed as configuration data inside a delimiter the file's own text cannot counterfeit, and permitted to add to the bundled rules but never to relax them. A team can therefore instruct its agents which elements and which path patterns concern a given repository, and a well-behaved agent's view narrows accordingly.

Deterministic identity is what makes the wider view coherent once repositories share a store. An element's identifier hashes only its normalized kind and name, never the author or the time, so two installations recording the same element arrive at the same node with nothing coordinated between them. The merged, cross-repository story exists because identity is computed rather than negotiated.

5. Advisory Filtering versus Enforcement

Everything in Section 4 is advisory, and this section exists to say so plainly. The store is plain newline-delimited JSON that any text tool can read and search without the engine. Anything that can read the repository, the store directory, or the synchronization bucket can read every decision in the store. There are no per-element permissions, and a prompt instructs a cooperating agent while placing no constraint on a hostile one. In the vocabulary of information flow control, kgai attaches no labels to decisions and mediates no reads [4, 5], and it issues no capability that could be granted per subgraph [6]. The confidentiality boundary is the substrate's boundary, filesystem permissions on the store directory and access policy on the bucket. Teams should provision both accordingly, because a scoping convention mistaken for access control is worse than no control at all [1].

What kgai does enforce, it enforces on identity and provenance rather than on reads. The committed configuration cannot take effect without a per-machine human approval, and it cannot name a synchronization destination at all. The store's configuration file is restricted to its owner with mode 0600 and holds a single installation identity, binding each store to a single user on a single machine, writes take a store-wide lock, and each installation writes its own shard and no other. Because identity is machine-bound, a copied store is caught and fails loudly rather than forking history in silence, and an intended copy receives a fresh identity through an explicit command. These mechanisms guarantee who wrote

what and prevent a configuration file from redirecting anyone's data. They do not, and are not claimed to, restrict who reads.

6. Monorepo and Multi-Repository Considerations

A monorepo gets coherent scoping by construction. One repository is one store under the default, and per-element and per-path recall are the instruments inside it. The considerations below concern the opposite arrangement, several repositories sharing one store.

A shared store has a single configuration and a single install identity, so every enrolled repository on a machine writes into the same shard. That is what makes the merged view work, and it is also why path conventions need care. A path property recorded as a bare source pattern is ambiguous once two repositories both contain a directory of that name, so the documentation recommends repository-qualified patterns in merged stores. The single-user constraint carries over unchanged. Several people on one machine must not share a store directory, since they would collide on permissions or write one shard from two processes. Each person keeps their own store, and synchronization merges the append-only shards without textual conflict, a per-writer arrangement in the spirit of conflict-free replication [7]. The trade-off profile, one shared instance serving several logical tenants in exchange for shared operational fate, is familiar from the multi-tenancy literature [8].

Migration into a shared store moves the log and nothing else. The shards are the store, every derived structure is rebuilt from them, so history is carried over by copying shard files and replaying them, with rationale, authorship, and original dates intact. The graph export exists for verification, and migration never passes through it. Leaving is asymmetric. A repository can return to its own store, but whatever it recorded into the shared store remains there, and the tooling says so rather than hiding it.

7. Possible Directions and Limitations

The roadmap lists optional decision signing for zero-trust team remotes. Signing addresses authenticity, whether a decision really came from the writer it names, and would strengthen the provenance guarantees of Section 5. It would not convert advisory read scoping into enforcement, since it protects the integrity of the log rather than the confidentiality of its contents. Genuine read enforcement, per-subgraph grants or mediated queries in the style of the reference monitors and capability systems of the classical literature [2, 6], would be a substantial departure from a local-first design built on plain files, and we name it here only as a possible direction. We promise neither.

The limitations are the mirror of the design. Scoping is advisory, so confidentiality planning must happen at the filesystem and bucket layer. Path and naming conventions in merged stores require discipline the tool cannot check for you. And the approval model trades convenience for consent, since every machine asks once, including the author's own.

8. Conclusion

kgai draws its default boundary at the repository and makes every widening of that boundary an explicit human act, a committed request that decides nothing until approved per machine, a remote that only the store itself can name. Inside a store, scoping is a query discipline, per-element recall and prompt conventions that pin a cooperating agent to a subgraph. The honest characterization of that discipline is advisory filtering. What kgai enforces is identity and provenance, one writer per shard, machine-bound identity, and a trust gate on configuration that arrives with a clone. What it does not enforce is reads, and anything that can read the repository can read the store. Teams that hold both halves of that sentence can share a decision graph across exactly the repositories that belong together, and no further.

References

[1] J. H. Saltzer and M. D. Schroeder. 1975. The Protection of Information in Computer Systems. Proceedings of the IEEE 63(9), 1278-1308.

[2] B. W. Lampson. 1974. Protection. ACM SIGOPS Operating Systems Review 8(1).

[3] E. Evans. 2003. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

[4] D. E. Denning. 1976. A Lattice Model of Secure Information Flow. Communications of the ACM 19(5).

[5] A. Sabelfeld and A. C. Myers. 2003. Language-Based Information-Flow Security. IEEE Journal on Selected Areas in Communications 21(1).

[6] J. B. Dennis and E. C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. Communications of the ACM 9(3).

[7] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. 2011. Conflict-free Replicated Data Types. Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011), Lecture Notes in Computer Science, Vol. 6976. Springer, 386-400.

[8] C. Bezemer and A. Zaidman. 2010. Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare? Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE-EVOL 2010).