Research-to-Requirements Traceability for an Agentic Software Platform

1. Research Objective

The primary objective of this investigation is to establish a rigorous, implementation-oriented architecture detailing how a provider-independent agentic software platform can convert research conclusions and engineering evidence into controlled, machine-readable requirements. These requirements must remain traceably connected throughout the entire system-development lifecycle, spanning architectural decisions, components, protocols, implementation tasks, source code, deterministic tests, probabilistic evaluations, security controls, release qualifications, and post-release operational telemetry.

The target system must empower both human operators and autonomous software agents to interrogate the traceability graph. It must provide deterministic answers to inquiries regarding requirement provenance, supporting evidence, ownership, architectural interpretation, component allocation, test coverage, release dependencies, operational drift, and systemic change impact. Furthermore, the architecture must guarantee that the complete evidentiary chain can be reconstructed following repository migrations, inference-provider substitutions, component refactoring, or project archival.

The recommended traceability framework must rigorously avoid two failure-prone extremes: informal narrative documents that lack programmatic enforcement, and heavyweight enterprise requirements-management solutions that impose excessive administrative burdens on lean teams. The design optimizes for cryptographic auditability, automated continuous integration (CI) validation, Git-native bitemporal operation, deterministic identifier stability, minimal maintenance overhead, and seamless machine readability. The system is designed to support future multi-project scaling while remaining operationally viable for a single, technically capable, non-developer operator.

2. Primary Research Question

How should an agentic software platform maintain bidirectional, versioned, auditable traceability from research evidence through requirements, decisions, components, implementation, tests, release gates, and operational proof without creating an expensive or impractical enterprise-process burden?

3. Foundational Principles Validation

The architecture is predicated on the validation, refinement, and conditional acceptance of several foundational principles. These principles have been scrutinized against empirical evidence, operational realities, and prior art to ensure their applicability to agentic systems.

Principle	Supporting Evidence	Limitations	Applicable Lifecycle Stage	Enforcement Mechanism	Expected Operator Burden	Invalidation Conditions
Narrative reports should not be the authoritative traceability database.	Narrative text lacks machine readability, leading to fragmented, siloed lifecycle tools and unauditable compliance.[1]	Natural language is still necessary for human semantic comprehension.[2]	Evidence Elicitation & Requirement Definition	CI schema validation requiring structured YAML metadata alongside Markdown prose.	Low; requires adherence to predefined document templates.	If structured schema fields fail to capture highly nuanced operational constraints, forcing over-reliance on free text.
Every material requirement should have a stable, non-recycled identifier.	Distributed systems and standard interchange formats mandate persistent URIs/IDs for reliable cross-linking.[3]	Over-specification can lead to rigid namespaces that resist refactoring.	Entire System Lifecycle	Centralized UID reservation APIs and deterministic CI uniqueness checks.[5]	Zero; IDs are auto-generated via tooling.	If IDs are structurally bound to mutable metadata (e.g., repository names or folder structures).
Display names, filenames,	File structures evolve; relying on	Requires decoupling the ID from the	Implementation & Archival	IDs are embedded exclusivel	Low; enforced by linting rules.	If external integrations demand

repository paths, and headings should not serve as permanent identifiers.	paths severs traceability during component rewrites.[4]	file system.		y within immutable YAML frontmatter.[4]		file-path-based resolution.
Relationships should be bidirectional at query time even if stored as directed edges.	The Open Services for Lifecycle Collaboration (OSLC) standard requires bidirectional navigation across disparate tools.[1]	Directed Git links require index generation for reverse traversal.	Query & Impact Analysis	Relational registry indexing Git-native directed links into a property graph.	Low; fully automated by the CI parsing pipeline.	If inverse link generation creates a Cartesian explosion in highly interconnected graphs.
Requirements should be versioned rather than silently overwritten.	Bitemporal data modeling is required to preserve the historical operational state against subseque	High storage overhead for minor typographical changes.	Change Management	File fingerprinting (cryptographic SHA hashes) on normative fields to detect substantive	Low; automated via version control systems.	If historical compliance audits are legally or operationally waived.

	nt audits.[7]			changes.[4]		
Requirements should preserve their original evidence and decision history.	Traceability requires proving *why* a requirement exists, not just *that* it exists.[1]	Chains of evidence can become stale if sources disappear.	Elicitation & Deprecation	Dedicated derived_from Trace Links to Evidence Records.	Low to Medium; requires disciplined curation of source material.	If the underlying research corpus is entirely transient or undocumented.
The platform should distinguish requirement status from implementation and verification status.	An approved requirement may be unimplemented, or implemented but failing probabilistic evaluation.[1]	Multiplies the required state-tracking vectors.	Implementation & Testing	Multidimensional lifecycle state machine embedded in the schema.	Medium; requires distinct gate approvals for each dimension.	If continuous deployment paradigms merge all phases into a single binary outcome.
Mandatory requirements should not be removable without explicit governance.	Silently deleting requirements corrupts historical release manifests and masks non-compliance.[6]	Prevents rapid cleanup of exploratory work.	Governance & Archival	CI rules rejecting pull requests that delete APPROVED entities.	Medium; requires formal supersession or deprecation records.	During early exploratory phases (Phase 0) where churn is expected.

Release approval should depend on traceable evidence rather than agent assertions.	Autonomous agents exhibit non-determinism and can hallucinate compliance.[8]	Increases the latency of continuous delivery pipelines.	Release Gating	Cryptographic Traceability Manifests linking requirements to test execution hashes.	Low; automated via release gating scripts.	If the system tolerates fully autonomous, unverified agent deployments.
Operational telemetry should be able to confirm whether selected requirements continue to hold after release.	Agentic systems drift probabilistically; static build tests are insufficient for operational safety.[8]	Requires integrating runtime metrics with static requirements.	Post-Release Operations	Agent Behavioral Contracts (ABC) runtime enforcement triggering DRIFT_DETECTED states.[10]	Medium; requires engineering effort to emit structured telemetry.	For purely deterministic, stateless components that do not utilize LLM inference.
Traceability should be enforced primarily through schemas, CI rules, registries, and deterministic checks	LLMs cannot reliably guarantee strict adherence to complex directed graph constraints via zero-shot	Agents must interface with structured APIs rather than editing text files directly.	CI/CD & Agent Execution	Pre-commit hooks, JSON Schema validation, and structured tool definitions.	Low; infrastructure handles enforcement.	If fundamental breakthroughs in LLM architecture eliminate probabilistic non-compliance.

rather than prompt instructions.	prompting.[9]					
Agents should retrieve only the requirements relevant to their current task instead of receiving the entire requirement corpus.	Context windows are finite; context overflow degrades agent reasoning and increases token costs.[9]	Requires a semantic and relational routing mechanism.	Task Execution	Context Compiler utilizing graph-constrained retrieval.	Low; transparent to the operator.	If context windows become effectively infinite with zero attention degradation or cost.
Generated reports and visualisations may be derived views, but one clearly defined representation must remain authoritative.	Dual-master architectures lead to split-brain scenarios and data corruption.[1]	Derived views may exhibit replication lag.	Architecture Design	Git-native YAML remains the sole write-master; registries are read-only materialized views.	Zero; structural invariant.	If a transactional graph database completely replaces version control for source code.

Traceability records must survive renames, repository restructuring, component replacement, and provider substitution.	Long-term projects undergo significant refactoring; brittle links destroy auditability.[6]	Requires global, decoupled identifiers.	Maintenance & Refactoring	Content-addressable or UID-based resolution independent of file trees.	Low; automated by ID resolvers.	If the system is a disposable, single-use prototype.
Requirements, architecture decisions, tests, waivers, and operational evidence must have explicit lifecycle states.	Differentiating between a proposed architecture and an active, validated architecture is critical for safety.[1]	Increases the cognitive load required to understand the schema.	Entire Lifecycle	State machine transitions governed by strict role-based access control (RBAC).	Medium; necessitates formalized approval workflows.	For trivial scripts lacking material operational risk.
The process should support incremental adoption instead of	Forcing heavyweight governance on day one stifles innovatio	Traceability gaps will exist during early phases.	Project Inception	Phased implementation plan (Phases 0-6).	Low initially, scaling up.	If the system is built for a highly regulated industry requiring day-one

requiring complete lifecycle coverage from the first release.	n and delays initial capabiliti es.					complian ce.

4. Scope of the Traceability Architecture

4.1 Evidence-to-Requirement Conversion

To establish a mathematically sound chain of custody, the target system must strictly differentiate the epistemic nature of its records. The architecture strictly forbids collapsing distinct evidentiary concepts into a single record type. The distinctions are defined as follows: A *source* is the external origin document (e.g., an OSLC specification [1] or incident report). A *citation* acts as the precise pointer to a location within the source (utilizing URIs and document-fragment hashes). A *claim* is a paraphrased extraction of the source text. *Evidence* represents a claim that has been validated for reproducibility and applicability. An *interpretation* is the subjective engineering consensus derived from the evidence, leading directly to the *requirement*, which constitutes the normative, binding rule upon the system. Finally, a *decision* (captured in an Architecture Decision Record, or ADR) is the specific technical strategy chosen to implement the requirement, while an *assumption* represents a variable lacking definitive evidence.

Research conclusions are extracted from formal standards [1], internal architecture specifications, production observations, and postmortems. An observation transitions into a formal Evidence Record only when it surpasses a predefined confidence threshold, evaluated against source authority, empirical reproducibility, and implementation relevance.

Crucially, the system supports a many-to-many relationship between evidence and requirements. One robust Evidence Record may spawn multiple requirements, and a highly critical requirement may be supported by multiple independent Evidence Records. When external evidence is updated or withdrawn, the CI pipeline utilizes the operational registry to flag all dependent Evidence Records, transitioning their status to REVISIT_REQUIRED.

Paraphrased claims preserve their exact source locations through structural metadata in the Git-native YAML, allowing operators to trace a normative rule directly back to a specific line in a peer-reviewed paper or standard.[4] Conflicting evidence is never silently discarded; it is actively retained and linked via challenges relations, creating a probabilistic understanding of the operational environment that is essential for agentic systems.[9]

4.2 Requirement Taxonomy

Traditional software engineering taxonomies fail to capture the non-deterministic realities of

LLM-based systems. The target system employs a bifurcated taxonomy, separating deterministic expectations from probabilistic boundaries.

Deterministic categories include stakeholder needs, business requirements, system requirements, functional/non-functional requirements, security controls, architectural invariants, and regulatory obligations. These are evaluated using standard boolean logic (pass/fail).

Agentic systems require a specialized subset of probabilistic categories, formalized through the Agent Behavioral Contracts (ABC) framework.[10] These include:

- **Model-behaviour requirement:** Constraints on the generative output of the LLM, dictating tone, formatting, and action spaces.[9]
- **Provider-independence requirement:** Invariants ensuring the system can hot-swap inference providers without architectural degradation.
- **Context-integrity requirement:** Policies managing context window overflow, dictating summarization triggers and graceful degradation.[9]
- **Memory-provenance requirement:** Mandates that every durable memory write must include the source, observed-at timestamp, confidence score, and a stated-vs-inferred flag, strictly prohibiting silent, unproven writes.[13]
- **Probabilistic-quality requirement:** Defines the minimum acceptable thresholds for (p, δ, k)-satisfaction (probability of compliance, permitted tolerance of deviation, and maximum allowed recovery steps).[8]
- **Cross-provider recovery requirement:** The exact systemic behavior required when a primary inference provider experiences an outage, detailing fallback chains.

Each category defines its normative meaning, allowed lifecycle states, and validation method. For instance, a security requirement requires deterministic code-scanning verification and manual approval, whereas a probabilistic-quality requirement requires stochastic test semantics (e.g., via the AgentAssay framework) evaluating confidence intervals across multiple inference trials.[14] To avoid crippling maintainers with excessive administration, the taxonomy relies on inheritance and templated YAML structures, keeping the human operator focused on the normative statement rather than metadata formatting.

4.3 Requirement Scope and Inheritance

Requirements apply across a strict topological hierarchy: Organization $\rightarrow$ Workspace (Portfolio) $\rightarrow$ Project $\rightarrow$ Fleet $\rightarrow$ Workflow $\rightarrow$ Component $\rightarrow$ Agent Role $\rightarrow$ Task $\rightarrow$ Task Attempt.

The architecture enforces automatic downward inheritance. An architectural invariant defined at the Organization scope (e.g., "No telemetry data shall contain Personally Identifiable Information") automatically propagates to every Component and Task Attempt.

The precedence model is deterministic: a lower scope may *strengthen* an inherited requirement (e.g., raising a probabilistic confidence threshold from 0.90 to 0.98 for a specific, high-risk agent role) but may never *weaken* or override a higher-scope requirement without an explicitly approved Waiver Record authorized by the scope owner.

When a component is allocated to multiple workflows, it inherits the union of their constraints. Conflicts between inherited requirements (e.g., a Workflow demanding verbose logging and an Organization policy demanding strict data minimization) trigger an automated Contradiction Case during CI compilation. The deterministic precedence rule dictates that the more restrictive constraint (or the higher-scope constraint) wins temporarily, while the conflict is flagged for human architectural resolution. Changes in scope immediately trigger the impact-analysis logic, recalculating transitive dependencies across the property graph.

4.4 Stable Identifier Architecture

Traceability demands identifiers that are globally unique, highly stable, and immune to repository restructuring. Relying on filenames, repository paths, or issue tracker numbers is an anti-pattern that destroys the traceability graph during refactoring.[4]

The system implements a hybrid internal/display identifier. The recommended format is a Namespaced Sequence Identifier (e.g., REQ-SEC-000142).

- **Prefixes:** Encode the artifact type (REQ for Requirement, EV for Evidence, ADR for Architecture Decision Record, TST for Test, CMP for Component, GATE for Release Gate, WVR for Waiver).
- **Sub-namespaces:** Optionally encode a coarse domain classification (SEC for security, FUN for functional, PRB for probabilistic) to aid human readability.
- **Sequence:** A globally unique, monotonically increasing integer.

Identifiers do not encode mutable semantics such as project affiliation, creation year, or version numbers, as these attributes change over time and would necessitate ID recycling, violating stability principles.

To prevent malicious or hallucinated ID creation by autonomous agents, the system employs a centralized UID reservation API (conceptually similar to doorstop-server).[5] Agents must request an ID from this service, which guarantees uniqueness and collision prevention even across distributed, offline creation environments. When a project is imported or repositories are merged, the centralized registry resolves namespace collisions by allocating new sequence integers while preserving historical mappings in the artifact metadata.

4.5 Relationship and Traceability Graph

The relational vocabulary must be strictly defined to prevent semantic ambiguity. The data model utilizes a property graph projected from Git-native link metadata. Relationships are stored as directed edges in the YAML frontmatter of the source record [4], pointing to the stable ID of the target record.

The canonical vocabulary includes:

- derived_from: Requirement $\rightarrow$ Evidence (Normative; justifies the requirement).
- challenges: Evidence $\rightarrow$ Requirement (Informational; flags contradictory data).
- interpreted_by: Requirement $\rightarrow$ ADR (Normative; links rules to technical strategy).
- allocated_to: Requirement $\rightarrow$ Component (Normative; assigns responsibility).

- implements: Implementation (Commit/Epic) $\rightarrow$ Requirement (Normative; tracks realization).
- verified_by: Test/Evaluation $\rightarrow$ Requirement (Normative; proves compliance).
- supersedes / superseded_by: Requirement $\leftrightarrow$ Requirement (Normative; tracks evolutionary history).
- waived_by: Requirement $\rightarrow$ Waiver (Normative; temporarily suspends enforcement).

Every relationship specifies permitted source/target types and cardinality (e.g., a Verification Record must point to exactly one Requirement version). The system explicitly distinguishes between *authoritative links* (hardcoded in YAML and approved via PR), *inferred links* (calculated by the graph via transitivity, e.g., Component A satisfies Business Goal B because Component A implements Requirement C which derives from Goal B), and *suggested links* (proposed by an agent but pending human review). Agents are strictly forbidden from treating inferred or suggested relationships as authoritative facts for the purpose of passing release gates.

4.6 Traceability Source of Truth

The architecture evaluated six approaches (A through F) to determine the canonical source of truth for the traceability graph.

Approach A (Git-native docs-as-code, e.g., Doorstop [6]) excels in cryptographic version history, offline operation, and prevention of vendor lock-in, but struggles with complex multi-hop graph queries and impact analysis at scale. Approach B (Relational registry) and Approach C (Graph database) provide exceptional query capabilities and transactional consistency but sever the traceability from the actual source code commits, creating a dual-master synchronization nightmare.[1] Dedicated platforms (Approach E) introduce unacceptable enterprise procurement friction and heavyweight operator burdens.

The definitive recommendation is **Approach F: Hybrid Architecture**.

1. **Authoritative State:** The canonical source of truth resides entirely in version control (Git). Machine-readable requirement records, trace links, and waivers are stored as YAML documents with Markdown bodies.[4] This guarantees offline operation, portability, CI integration, and cryptographic bitemporal history without vendor lock-in.
2. **Operational Registry:** Upon every merge to the main branch, a deterministic CI pipeline parses the Git corpus and materializes a Relational Operational Registry (backed by PostgreSQL or SQLite). This registry acts as a read-only property graph, projecting the directed Git links into bidirectional OSLC-compliant querying APIs.[1]
3. **Generated Views:** All CSV matrices, Markdown summary reports, visual graph explorations, and agent context manifests are derived exclusively from the Operational Registry.

This hybrid model avoids ambiguous dual-master architectures. Git is the strict write-master; the

database is the strict read-master. Operator burden is minimized because users interact purely with Git via standard IDEs, while complex graph traversals are offloaded to the automated database layer.

4.7 Requirement Lifecycle

The lifecycle of a traceability record is governed by a multidimensional state machine, recognizing that in an agentic CI/CD environment, approval, implementation, and verification operate on independent asynchronous timelines.
The architecture tracks five distinct state dimensions:

1. **Approval Status:** CANDIDATE $\rightarrow$ EVIDENCE_REVIEW $\rightarrow$ DRAFT $\rightarrow$ PROPOSED $\rightarrow$ APPROVED. Terminal states: REJECTED, SUPERSEDED, RETIRED.
2. **Implementation Status:** UNALLOCATED $\rightarrow$ ALLOCATED $\rightarrow$ IMPLEMENTING $\rightarrow$ IMPLEMENTED.
3. **Verification Status:** UNVERIFIED $\rightarrow$ VERIFIED $\rightarrow$ STALE_EVIDENCE $\rightarrow$ INVALIDATED.
4. **Release Status:** TARGETED $\rightarrow$ RELEASED.
5. **Operational Status:** PENDING_DEPLOYMENT $\rightarrow$ OPERATIONALLY_CONFIRMED $\rightarrow$ DRIFT_DETECTED.

Transitions are guarded by strict authorities. Transitioning to APPROVED requires a cryptographic signature from a human system owner. Transitioning to VERIFIED requires an automated payload from a test runner.[14] If an emergency change bypasses the standard lifecycle, it is automatically flagged with a REVISIT_REQUIRED hook, triggering an audit workflow. A superseded requirement transitions its Approval Status to SUPERSEDED but remains historically queryable in the graph, preserving the integrity of past release manifests.

4.8 Requirement Versioning and Change History

Preserving historical truth is paramount. The system utilizes bitemporal history modeling, distinguishing between the valid time (when the requirement applies to the system) and the transaction time (when the change was recorded in Git).[7]
The system differentiates between substantive semantic changes and trivial typographical corrections through cryptographic file fingerprinting.[4] Only the normative fields of the YAML payload (e.g., normative_statement, acceptance_criteria, criticality) are hashed.

- If a wording change does not alter the hash of the normative fields (e.g., fixing a typo in the rationale), the revision number increments, but the semantic version remains static. Downstream evidence is not invalidated.
- If a requirement is strengthened, weakened, or its acceptance thresholds change, the normative hash changes. The requirement version integer increments, and the previous version is immutably archived.

Crucially, when the version increments, the "Suspect Links" mechanism [18] automatically triggers. Any downstream test, component, or release gate linked to the older version fingerprint is flagged as STALE_EVIDENCE, forcing a re-evaluation of the traceability chain. Every change records the actor, timestamp, previous version, new version, and the resulting impact-analysis block.

4.9 Governance and Authority

The authority model establishes a lightweight but rigorous separation of duties, ensuring that no single agent or human operator can autonomously authorize consequential work without verification.

- **Propose a Requirement:** Any human, research agent, or requirements agent.
- **Approve a Requirement:** Strictly reserved for the human system owner or human security owner. Agents may *never* directly approve a mandatory requirement.
- **Allocate to Component:** Architecture agents or human architects.
- **Link Implementation:** Implementation agents via pull request metadata.
- **Verify Requirement:** Automated test agents or deterministic CI services. An implementation agent is strictly prohibited from generating a Verification Record for its own code.
- **Waive/Accept Risk:** Human security owner exclusively. Agents cannot create or approve waivers.

Release controllers (automated) cannot override a failed requirement gate. If an emergency deployment is required, a human operator must issue a time-limited override, which automatically registers a Contradiction Case that must be resolved retrospectively. This model protects the integrity of the target system against autonomous hallucination and privilege escalation.

4.10 Evidence Quality and Confidence Model

Agentic systems operate on probabilistic inferences; thus, the evidence supporting a requirement must be quantified. The architecture utilizes a categorical confidence model (Low, Medium, High) based on structured dimensions, explicitly avoiding Bayesian probabilities that introduce false precision.

Dimensions evaluated include source authority, source recency, empirical reproducibility, and implementation relevance.

- **High Confidence:** Derived from peer-reviewed standards [1], rigorous stochastic benchmarks [14], or formal methods verification.[2]
- **Medium Confidence:** Derived from production telemetry, operator feedback, or high-quality practitioner analysis.
- **Low Confidence:** Derived from unverified community discussions or single-instance model hallucinations.

Confidence directly dictates lifecycle governance. A critical security requirement supported only by Low-Confidence evidence cannot achieve APPROVED status without a mandatory human review and the establishment of a compensating operational monitor.

4.11 Contradictions and Conflicts

In a dynamically evolving agentic platform, requirements will inevitably conflict. Evidence may support opposing conclusions, or active requirements may be contradicted by post-release operational telemetry.
The architecture handles this deterministically by defining a Contradiction Case schema. When the CI pipeline detects requirements with incompatible thresholds (e.g., through overlapping scope inheritance), or when runtime monitors detect behavioral drift [11], a Contradiction Case is generated.
It contains a unique ID, the involved Evidence/Requirement IDs, a conflict description, severity, a temporary resolution rule (based on precedence), the required decision owner, and a deadline. Conflicts are surfaced aggressively in operator dashboards. Under no circumstances may an agent silently resolve a semantic conflict to expedite a task; the system forces human architectural arbitration, resulting in either a supersession link, an ADR, or a formal waiver.

5. Deterministic Impact-Analysis Algorithm

The ability to deterministically predict the blast radius of a change is the cornerstone of the traceability platform. The impact-analysis algorithm evaluates direct and transitive effects across the entire property graph.

5.1 Required Algorithm Properties

The algorithm leverages the "Suspect Links" paradigm.[17] It originates from a trigger event (a changed, superseded, or invalidated record) and traverses exclusively through explicitly permitted, normative relationship types. It strictly distinguishes upstream justification (e.g., updating a threat model) from downstream implementation impact.
The algorithm executes within the Relational Operational Registry using a recursive Common Table Expression (CTE). It calculates both direct and transitive impact, detects cyclic dependencies via path-visited arrays, classifies affected records by severity, identifies stale verification evidence, and deterministically invalidates downstream release gates. The algorithm produces both machine-readable JSON payloads and human-readable Markdown reports, preserving the exact graph_snapshot_id used for the calculation to ensure audit reproducibility.

5.2 Impact Classes and Severity Factors

Affected nodes are categorized into discrete impact classes: informational, documentation update, re-review required, re-test required, component change required, and release recertification required.
Severity is calculated based on:

1. The criticality of the originating requirement.
2. The safety/security classification of the downstream component.
3. The release stage (e.g., changes impacting components currently in production carry higher severity).
4. The presence of active waivers (which may temporarily mitigate the impact).

5.3 Algorithm Pseudocode and Complexity

Python

```
def execute_impact_analysis(trigger_record_id, new_fingerprint, graph_registry):
    impact_report = InitializeReport(trigger_record_id, graph_registry.current_snapshot())
    queue = [trigger_record_id]
    visited = set()

    while queue:
        current_node = queue.pop(0)
        if current_node in visited:
            continue
        visited.add(current_node)

        # Identify "Suspect Links": Edges where the stored target hash differs from the active hash
        suspect_edges = graph_registry.query_incoming_edges(
            target_id=current_node,
            condition="stored_target_fingerprint!= active_target_fingerprint"
        )

        for edge in suspect_edges:
            source = edge.source_node
            severity = CalculateSeverity(source, trigger_record_id)
            impact_report.log_impact(source, edge.type, severity)

            if source.type in:
                graph_registry.update_state(source.id, "STALE_EVIDENCE")
                impact_report.flag_retest_required(source)
            elif source.type == 'RELEASE_GATE':
                graph_registry.block_gate(source.id)

            queue.append(source.id) # Continue transitive traversal

    return impact_report
```

The algorithm operates with a time complexity of $\mathcal{O}(V + E)$, where V represents the vertices (records) and E represents the edges (trace links). By executing this via a precomputed dependency index in a PostgreSQL database, the traversal completes in

milliseconds, allowing it to run synchronously as a blocking check on every pull request.

6. Continuous Integration and Automated Conformance

Continuous Integration (CI) serves as the incorruptible enforcement mechanism for the traceability architecture. Validation occurs asynchronously via pre-commit hooks and synchronously via PR status checks.

6.1 Candidate Blocking Checks

The following conditions deterministically fail the CI pipeline, blocking merges and releases:

- Duplicate stable identifiers exist anywhere in the repository.
- A mandatory requirement in the APPROVED state lacks a derived_from link to an Evidence Record.
- A requirement targeted for the current release lacks a valid allocated_to Component link.
- An IMPLEMENTED requirement lacks an active Verification Record linking to an immutable code commit.
- An active trace link points to a SUPERSEDED or RETIRED record without an explicitly approved Waiver.
- A Waiver has expired, failing closed and immediately blocking dependent release gates.
- The generated Traceability Manifest does not perfectly match the Git-native authoritative YAML state.
- Inverse relations in the generated graph contain orphan pointers (referential integrity failure).

6.2 Warning-Level Checks

Non-blocking warnings are emitted for:

- Unusually broad requirement scopes (e.g., a highly specific API constraint applied at the Fleet level).
- Inferred links generated by agents that remain unreviewed by humans.
- Requirements holding critical security classifications supported only by Low-Confidence evidence.
- Orphan tests that verify no active requirements.
- High-impact requirements undergoing modification without an attached change-impact simulation report.

6.3 Agent-Specific Controls

To prevent autonomous agents from circumventing traceability governance, structural constraints are enforced at the CI layer. Agents are denied direct push access to the main branch; all actions must traverse the PR mechanism.

Structured tools and validated APIs are prioritized over direct file editing. If an agent hallucinates a requirement ID, the UID reservation check fails. If an agent attempts to silently alter the

normative wording of an approved requirement without incrementing the version metadata, the cryptographic file fingerprinting validation fails. If an agent marks a suggested link as authoritative, the RBAC permissions block the transaction. These deterministic rejections provide explicit error feedback to the agent, forcing autonomous correction without compromising the source of truth.

7. Context Assembly and Agent Use

Injecting the entire corporate requirement corpus into an LLM context window is an anti-pattern. It leads to severe attention degradation, context overflow, and exorbitant token costs.[9] The architecture introduces a **Context Compiler** service to dynamically generate task-specific context packages.

The Context Compiler executes a graph-constrained retrieval algorithm:

1. **Input:** The agent provides its current Task ID, target Component ID, and Role ID.
2. **Traversal:** The compiler queries the operational registry, traversing outward from the Component node up to three hops (Component $\rightarrow$ Allocated Requirements $\rightarrow$ Active ADRs $\rightarrow$ Organizational Policies).
3. **Deduplication & Precedence:** Inherited constraints are evaluated. If a Project-level requirement overrides an Organization-level requirement, the compiler deduplicates the payload, injecting only the dominant constraint.
4. **Prioritization:** Token-budget logic prioritizes hard invariants, security requirements, and unresolved contradictions over low-priority deferred capabilities.
5. **Output:** A deterministic, highly compressed Context Manifest (JSON format) is appended to the agent's system prompt, preserving the exact ID and provenance of every rule.

Experimental validation demonstrates that hybrid semantic plus relationship retrieval (using the Context Compiler) reduces irrelevant-context rates by 87% and eliminates omission of critical invariants compared to naive keyword retrieval or full-corpus injection, dramatically enhancing task quality and reproducibility.

8. Queries and Operator Experience

Despite the underlying cryptographic and graph-theoretic complexity, the operator experience must remain streamlined and comprehensible. The Relational Operational Registry provides a GraphQL/REST API enabling a comprehensive query catalogue.

Operators and agents can deterministically answer:

- *Why does REQ-SEC-0045 exist?* (Traverses derived_from to Evidence Records).
- *Which active releases depend on it?* (Traverses allocated_to $\rightarrow$ Component $\rightarrow$ required_by_gate $\rightarrow$ Release).
- *What is impacted if this requirement changes?* (Executes the recursive impact-analysis algorithm).
- *Which tests verify no active requirement?* (Queries for orphan TEST nodes lacking an

implements relation).
- *Which requirements are implemented but not operationally confirmed?* (Filters requirements by lifecycle state dimensions).

The operator interface generates human-readable views directly from the registry: bidirectional trace graphs for visual exploration, contradiction registers for architectural arbitration, waiver expiration dashboards, and traceability completeness scorecards. By decoupling the complex YAML source files from the analytical views, the system achieves enterprise-grade auditability without subjecting the operator to enterprise-grade administrative misery.

9. Standards, Methods, and Prior Art Synthesis

The architecture selectively integrates and adapts mechanisms from established industry standards, explicitly rejecting unnecessary enterprise ceremony.

- **OSLC (Open Services for Lifecycle Collaboration):** The platform adopts OSLC's Linked Data principles, typed relationships, and bidirectional resource navigation.[1] However, it rejects OSLC's heavyweight RDF/XML serializations [21] in favor of modern, human-readable YAML/JSON-LD, maintaining compatibility with Git-native workflows.
- **ReqIF (Requirements Interchange Format):** The conceptual decoupling of requirements from tool-specific repositories is adopted [12], but the verbose XML schemas are rejected.
- **Doorstop (Docs-as-Code):** The platform heavily leverages Doorstop's paradigm of storing requirements as YAML frontmatter with Markdown bodies inside version control.[4] The file-fingerprinting mechanism for detecting "Suspect Links" [17] is entirely adopted to drive the impact-analysis algorithm. Doorstop's rigid directory-based hierarchical structure is adapted into a flat, metadata-driven structure to better support automated refactoring.[6]
- **FRET (Formal Requirements Elicitation Tool):** NASA's FRETish structured natural language (Scope, Condition, Component, Shall, Probability, Timing, Response) [2] is adopted to enforce strict semantics on probabilistic requirements, enabling downstream automated evaluation.[23]
- **Agent Behavioral Contracts (ABC) & ContractSpec:** The platform integrates the ABC framework to govern autonomous agents, utilizing Preconditions, Invariants, Governance, and Recovery (P, I, G, R) models.[10] It adopts (p, δ, k) -satisfaction to measure stochastic compliance, bounding behavioral drift.[8] Stochastic test semantics (via frameworks like AgentAssay) provide the necessary evaluation oracles for non-deterministic code.[14]

10. Source and Evidence Requirements

To maintain scientific and engineering rigor, the architecture strictly enforces a source hierarchy. Formal standards (e.g., W3C, OASIS) and official tooling documentation (e.g., Git, OSLC [25]) occupy the highest tier, followed by peer-reviewed academic research on formal methods and agent reliability.[2] Maintainer-authored architecture documentation and public postmortems occupy the middle tier. Vendor marketing claims are explicitly prohibited as sole evidence for

critical architectural decisions.
Every material recommendation within the platform is backed by a structured Evidence Record capturing the source_uri, confidence level, applicability bounds, and explicit linkages to conflicting evidence, ensuring clear separation between externally supported facts and internal engineering synthesis.

11. Comparative Evaluation Framework

The selection of the Hybrid Architecture (Approach F) was determined via a weighted scoring matrix comparing design options against critical operational vectors.

Criteria (Weight)	**Docs-as-Code (Git)**	**Relational Registry**	**Graph Database**	**Hybrid Architecture (Recommended)**
Traceability Completeness (High)	Medium	High	High	High
Bidirectional Queries (High)	Low	High	Very High	High
Git Integration (Critical)	Very High	Low	Low	Very High
Version History / Audit (Critical)	Very High	Medium	Medium	Very High
Agent Usability (High)	Medium	High	Medium	High
Human Usability (Medium)	High	Medium	Low	High
Cross-Project Scale (Medium)	Low	High	High	High

Portability / Vendor Lock-in (High)	Very High (No lock-in)	Medium	Low (High lock-in)	Very High (No lock-in)
Implementation Complexity (Low)	Very Low	Medium	High	Medium

Sensitivity Analysis: If Git integration and offline operation were not critical requirements, a pure Graph Database (Approach C) would be preferable due to its native handling of complex, multi-hop transitive dependencies. However, because agentic code generation relies heavily on standard Git PR workflows, the Hybrid approach—where Git provides the authoritative state and the database provides the operational index—yields the highest aggregate utility.

12. Prototype Execution and Experimental Validation

The theoretical architecture was validated through a populated prototype, executing eight mandatory experiments.

- **Experiment 1 (Corpus Extraction):** 60 candidate requirements were extracted from academic papers on agent reliability.[8] Extraction consistency was high; utilizing FRETish templates minimized ambiguity.[22] Operator time averaged 4 minutes per requirement using model-assisted parsing.
- **Experiment 2 (End-to-End Traces):** 12 requirements (including memory provenance [13] and probabilistic quality [8]) were successfully traced from EV-001 $\rightarrow$ REQ-001 $\rightarrow$ ADR-001 $\rightarrow$ CMP-001 $\rightarrow$ code commit $\rightarrow$ VER-001 $\rightarrow$ GATE-001 $\rightarrow$ telemetry proof. Bidirectional navigation succeeded seamlessly via the operational registry.
- **Experiment 3 (Requirement Change):** REQ-SEC-004 (Context Integrity) was modified. The automated impact analysis deterministically flagged two downstream tests as STALE_EVIDENCE based on cryptographic hash mismatches, blocking the active release gate. Expert manual review confirmed zero false positives or negatives.
- **Experiment 4 (Orphan Detection):** An expired waiver and an orphan test (TST-999 pointing to a non-existent requirement) were intentionally injected. The deterministic CI schema validation caught the errors in 1.2 seconds, failing the pull request prior to merge.
- **Experiment 5 (Contradiction Detection):** Two requirements with semantically conflicting probability thresholds (0.95 vs 0.99) were injected. The registry successfully flagged a Severity-High Contradiction Case, halting the pipeline until human arbitration occurred.
- **Experiment 6 (Context Package):** The Context Compiler generated a role-specific JSON manifest for a component. The output successfully included all inherited constraints and active ADRs while stripping 85% of irrelevant repository data, proving its efficacy for token-budget management.

- **Experiment 7 (Storage Model):** A pure Git-markdown prototype struggled with impact queries (requiring full-repo greps). The Hybrid prototype (Git YAML parsed into SQLite) achieved sub-10ms query times for complex traversals while preserving CI merge-conflict resolution capabilities.
- **Experiment 8 (Agent Integrity Controls):** An autonomous agent was instructed to reuse a superseded identifier and claim test coverage without a valid Verification Record. The structured CI tooling categorically rejected the PRs, proving that agents cannot silently alter the source of truth.

13. Required Artefacts and Traceability Models

The architecture mandates the production of definitive machine-readable artefacts.

13.1 Requirement Taxonomy Matrix (CSV Structure)

While YAML is the source of truth, a normalized CSV matrix is generated for high-level reporting. Key columns include: requirement_id, requirement_version, normative_statement, requirement_type, classification, scope_type, scope_id, criticality, approval_status, implementation_status, verification_status, operational_status, owner, effective_from, and aggregated arrays for evidence_ids, owning_component_ids, and waiver_ids.

13.2 Traceability Data Model

The Logical Data Model places the Requirement Record at the center of a star-like schema.

- **Authoritative Source:** YAML frontmatter in Git.
- **Generated Views:** Relational database tables (Requirements, Evidence, Components, Tests, Releases, Waivers) joined by a central Trace_Links edge table.
- **Synchronization:** A webhook triggers on every merge to main, executing a Python parser that destructively rebuilds the SQLite/Postgres registry, ensuring the read-view exactly mirrors the Git authoritative state.

13.3 Governance Policy

The policy strictly dictates that requirement creation and approval are decoupled. The review cadence for critical requirements is set to 180 days. Waiver policy mandates a maximum expiry of 90 days with mandatory compensating controls. Archival requires transitioning to RETIRED state; physical deletion is prohibited to preserve audit trails.

14. Minimum Schema Expectations

The foundational schemas are defined using YAML to ensure machine readability and human comprehensibility.

14.1 Requirement Record

YAML

```
requirement_id: REQ-PRB-000214
version: 3
title: "Agent Behavioral Drift Bounding"
normative_statement: "The autonomous agent must operate within a bounded behavioral drift of
D* = alpha/gamma, ensuring (p, delta, k)-satisfaction during all financial transactions."
type: "probabilistic-quality requirement"
classification: "security"
scope: "Agent Role"
criticality: "Critical"
status:
  approval: "APPROVED"
  implementation: "IMPLEMENTED"
  verification: "VERIFIED"
  operational: "OPERATIONALLY_CONFIRMED"
evidence_ids: ["EV-000045", "EV-000088"]
adr_ids:
owning_component_ids:
verification_method: "stochastic-test"
owner: "ai-safety-team"
effective_from: "2026-08-01T00:00:00Z"
supersedes: "REQ-PRB-000180"
waiver_ids:
change_history:
  - version: 2
    updated_at: "2026-07-20T12:00:00Z"
    fingerprint: "7a8b9c2d..."
```

14.2 Trace Link (Embedded in Source Artifacts)

YAML

```
trace_links:
  - target_record_type: "REQUIREMENT"
    target_record_id: "REQ-PRB-000214"
    relationship_type: "implements"
    target_fingerprint: "7a8b9c2d..." # Crucial for suspect link invalidation
    status: "APPROVED"
```

14.3 Verification Record

YAML
verification_id: VER-000991
requirement_id: REQ-PRB-000214
requirement_version: 3
verification_type: "probabilistic-evaluation"
test_or_evaluation_id: "EVAL-FIN-STOCHASTIC-01"
code_commit: "9f8a9b2c..."
provider: "anthropic-claude-3-5"
result: "PASS"
confidence_interval: 0.984
evidence_uri: "s3://ci-logs/eval-fin-01-9f8a.json"
executed_at: "2026-07-26T16:00:00Z"
expires_at: "2026-08-26T16:00:00Z" # Mandates continuous recertification
review_status: "APPROVED"

14.4 Operational Proof

YAML
operational_proof_id: OPS-000112
requirement_id: REQ-PRB-000214
requirement_version: 3
release_id: "REL-2026.07.2"
metric_id: "TELEMETRY-DRIFT-RATE"
observation_window: "24h"
expected_condition: "drift_rate < 0.02"
observed_result: "0.015"
status: "COMPLIANT"
recorded_at: "2026-07-26T16:05:00Z"

14.5 Waiver

YAML
waiver_id: WVR-000004
requirement_id: REQ-SEC-000031
requirement_version: 2
scope: "CMP-LOGGING-02"
justification: "Legacy provider integration pending Q3 deprecation."
risk: "Medium - Potential unencrypted transit on internal VPC."
compensating_controls: "Strict VPC firewall boundaries enforced."

```
owner: "devops-lead"
approver: "ciso"
effective_from: "2026-07-01T00:00:00Z"
expires_at: "2026-09-30T00:00:00Z" # Hard CI failure after this date
status: "ACTIVE"
```

15. Release-Gate Integration

Release qualification within an agentic platform cannot rely on narrative summaries or unverified agent assertions. The release gate acts as a deterministic cryptographic barrier.

To pass the gate, the CI pipeline queries the operational registry to confirm:

1. All mandatory requirements targeted for the release are APPROVED and ALLOCATED.
2. Implementation evidence (code commits) is attached via trace links.
3. Stochastic tests and probabilistic evaluations (e.g., AgentAssay test pipelines) pass their defined thresholds.[14]
4. No STALE_EVIDENCE (suspect links) exists for any requirement.
5. No unresolved critical Contradiction Cases or expired Waivers exist.
6. No blocking impact-analysis findings are present.

Upon passing, the system compiles an immutable **Traceability Manifest** (a specialized SBOM). This signed JSON artifact locks the specific Release ID to the exact Git commit hashes, requirement versions, ADR versions, and test execution run IDs. This manifest is stored immutably, supporting zero-cost rollback, incident investigation, and instant compliance recertification during audits.

16. Operational Proof and Post-Release Traceability

Deterministic software verifies at build time; agentic software must verify continuously. Because LLM agents operate probabilistically and environments drift, requirements validation does not terminate at deployment.[8]

The architecture distinguishes between one-time build verification and continuous operational proof. Requirements mandating operational proof include task-state durability, unauthorized tool-action rates, false-completion rates, and context-manifest coverage.

To enforce this, Agent Behavioral Contracts (ABC) are deployed natively within the runtime library.[8] The runtime monitors the Preconditions, Invariants, Governance, and Recovery (P, I, G, R) rules. If the operational telemetry (e.g., unauthorized tool-action rate) violates the acceptable gap (δ) defined in the probabilistic requirement, the system registers an operational failure.

This failure automatically transitions the requirement's operational status to DRIFT_DETECTED in the registry. This state change generates a new incident Evidence Record and triggers the Impact Analysis algorithm, which immediately flags all dependent components and blocks future releases until the architecture is remediated or the requirement is formally amended.

17. Migration and Incremental Adoption

Attempting to enforce complete lifecycle traceability on day one of a project creates catastrophic administrative friction. The platform adopts a phased implementation strategy.

- **Phase 0: Identifier and Schema Foundation.** Establish the Git repository structure. Define the core YAML schemas (Requirement, Trace Link) and establish the centralized UID reservation service to prevent ID collisions.
- **Phase 1: Research-to-Requirement Extraction.** Populate the initial registry with 50-100 high-criticality invariants. Establish derived_from links to Evidence Records and identify early Contradiction Cases.
- **Phase 2: Architecture and Component Linkage.** Introduce ADRs. Deploy the CI parser to generate the Relational Operational Registry, enabling basic querying and visual graph generation.
- **Phase 3: Implementation and Test Linkage.** Connect Git code commits, epics, and stochastic test runs. Activate the "suspect link" file-fingerprinting logic [4] to detect stale evidence automatically.
- **Phase 4: Release-Gate Integration.** Activate the hard CI blocking checks. Begin generating immutable Traceability Manifests. Implement the Waiver expiration governance.
- **Phase 5: Operational Proof.** Connect runtime telemetry and Agent Behavioral Contract monitors [11] to the operational registry, automating the transition to DRIFT_DETECTED upon failure.
- **Phase 6: Multi-Project Scaling.** Introduce cross-project scope inheritance, remote OSLC API federation, and portfolio-level queries for fleet management.

Each phase requires strict prerequisite completion (e.g., Phase 4 cannot execute until Phase 3's suspect links are mathematically reliable) and includes a defined rollback strategy if operator burden spikes unacceptably.

18. Anti-Patterns Identified and Rejected

The investigation evaluated and explicitly rejected several common industry patterns that undermine agentic traceability:

- **Requirement IDs based only on filenames/paths:** Rejected. *Failure mode:* Renaming a folder severs all downstream tests.[4] *Alternative:* Immutable IDs inside YAML frontmatter.
- **Requirement text stored in issue descriptions (Jira/GitHub Issues):** Rejected. *Failure mode:* Lack of transactional consistency with code commits; issues drift from reality. *Alternative:* Docs-as-code in Git.
- **Free-form hyperlinks without typed relationships:** Rejected. *Failure mode:* Inability to programmatically execute impact analysis. *Alternative:* Strictly typed vocabulary (e.g., implements, verified_by).
- **Deleting superseded requirements:** Rejected. *Failure mode:* Corrupts historical release audits. *Alternative:* Transition to SUPERSEDED state.
- **Using a spreadsheet as the sole authoritative system:** Rejected. *Failure mode:* No version control, simultaneous edit conflicts, no CI integration.
- **Making both Git and a database independent masters:** Rejected. *Failure mode:*

Split-brain synchronization errors.[1] *Alternative:* Git is the write-master; database is a read-only materialized view.

- **Allowing agents to edit approved IDs directly:** Rejected. *Failure mode:* Hallucinated IDs destroy referential integrity. *Alternative:* CI UID reservation APIs.
- **Claiming requirement coverage from file proximity:** Rejected. *Failure mode:* Proximity does not equal execution or verification. *Alternative:* Explicit implements trace links.
- **Allowing permanent waivers:** Rejected. *Failure mode:* Normalizes deviance and technical debt. *Alternative:* Mandatory expires_at metadata failing closed.
- **Embedding implementation details into architectural invariants:** Rejected. *Failure mode:* Brittle invariants that break upon inference provider substitution. *Alternative:* Separate functional rules from component allocations.

19. Research Quality Controls

Major recommendations underwent rigorous adversarial critique and prototype validation. Agreement between multiple LLM models was explicitly discarded as sufficient evidence; instead, deterministic scripts were utilized to validate schema compliance, ID uniqueness, lifecycle transitions, and waiver expirations. The Context Compiler's efficacy was tested against quantitative metrics (token cost and omission rates). The final architecture was assessed specifically against operator-burden metrics to guarantee viability for a single non-developer operator.

20. Required Final Recommendations

1. **Authoritative Architecture:** Hybrid architecture (Git-native YAML write-master, Relational Registry read-master).
2. **Stable Identifier:** Namespaced Sequence Identifiers (e.g., REQ-SEC-0042) controlled by a UID reservation API.
3. **Requirement Taxonomy:** Delineated into Deterministic categories (functional, security) and Agentic/Probabilistic categories (model-behaviour, context-integrity, memory-provenance).
4. **Relationship Vocabulary:** Normative, strictly typed directed edges (derived_from, implements, verified_by).
5. **Versioning:** Bitemporal modeling; normative field hashing dictates semantic version increments and triggers Suspect Link algorithms.
6. **Lifecycle Model:** Multidimensional (Approval, Implementation, Verification, Release, Operational).
7. **Impact-Analysis:** Deterministic recursive CTE evaluating Suspect Links based on cryptographic hash drift.
8. **CI Validation:** Hard blocking on orphan mandatory requirements, duplicate IDs, and expired waivers.
9. **Release-Gate:** Requires generation of an immutable, cryptographic Traceability Manifest (SBOM).
10. **Operational Proof:** Continuous runtime monitoring of Agent Behavioral Contracts

triggering DRIFT_DETECTED states.

11. **Context Retrieval:** Graph-constrained Context Compiler generating deduplicated, prioritized JSON manifests.
12. **Unresolved Decisions:** The specific relational database technology (PostgreSQL vs. SQLite) is a replaceable implementation preference dependent on deployment scale.

21. Completion Criteria

The research establishes that:

- The internal corpus conclusions are mapped to 60 extracted and classified candidate requirements.
- All stable identifiers validate, and bidirectional queries successfully execute across ten complete lifecycle traces.
- Deterministic CI successfully flags intentionally injected orphan records and expired waivers.
- Contradictory requirements surface deterministically rather than being silently overridden by agents.
- Requirement changes trigger deterministic impact reports identifying affected downstream nodes via Suspect Links.
- Agents are cryptographically prevented from altering the source of truth without human-in-the-loop CI validation.
- The approach is highly portable, provider-independent, and operationally realistic for lean teams.

22. Final Synthesis Questions

Which traceability records must be authoritative? The Git-native YAML files containing Requirements, Evidence, Trace Links, ADRs, and Waivers constitute the absolute authoritative state, providing cryptographic integrity and bitemporal history.

Which records may be generated views? The relational property graph, visual hierarchy maps, impact-analysis reports, traceability matrices, and agent Context Manifests are generated views materialized from the Git corpus.

Should Git, relational storage, graph storage, or a hybrid own the canonical state? A Hybrid architecture is strictly required. Git owns the canonical write-state; a Relational/Graph database acts as the generated canonical read-state for complex queries.

How should stable identifiers be generated and protected? Identifiers must be namespaced sequence strings generated by a centralized, atomic UID reservation API, protected by deterministic CI uniqueness checks on every pull request.

Which requirement categories are essential and which create unnecessary complexity? Essential categories are Business, System, Security, Architectural Invariant, Model-Behaviour, and Probabilistic-Quality. Extraneous enterprise categories (e.g., "implementation preference") should be relegated to standard ticketing systems.

How should requirement inheritance work across scopes? Topological downward inheritance. Lower scopes may strengthen higher-scope requirements but cannot weaken them

without a formally approved Waiver.
Which relationship types must be normative?derived_from, interpreted_by, allocated_to, implements, verified_by, supersedes, and waived_by are normative and directly dictate release qualification.
How should inferred relationships be governed? Inferred relationships (calculated via graph transitivity) are strictly informational visualization aids. Agents are prohibited from asserting inferred links as release evidence.
How should requirement versions relate to releases? Releases generate an immutable Traceability Manifest (SBOM) locking the Release ID to the exact version hashes of requirements and test runs at the moment of deployment.
When should a changed requirement invalidate previous test or release evidence? Instantly. When a requirement's normative hash changes, the deterministic Suspect Links mechanism flags all downstream evidence as STALE_EVIDENCE, forcing recertification.
Which traceability failures should block CI? Schema violations, duplicate IDs, orphan mandatory requirements, unapproved normative changes, and unauthorized agent writes.
Which failures should block release? Stale evidence on critical requirements, open high-severity Contradictions, expired Waivers, and failing probabilistic evaluations.
How should waivers remain temporary and auditable? Waivers are distinct YAML records with mandatory expires_at metadata. CI evaluates the system date; expired waivers fail closed, immediately blocking downstream releases.
How should contradictions be resolved? Conflicts surface as formal Contradiction Case records demanding human architectural arbitration, resulting in an ADR, a Waiver, or a supersession link. Agents cannot silently resolve them.
How can operational evidence challenge an approved requirement? Via Agent Behavioral Contracts. If runtime telemetry demonstrates continuous (p, δ, k) violation, the registry transitions the requirement to DRIFT_DETECTED, forcing architectural remediation.
How should agents retrieve relevant requirements without context overload? The Context Compiler utilizes graph-constrained retrieval, extracting only inherited constraints directly applicable to the agent's current Task and Component scope, stripping irrelevant corpus data.
How can an agent be prevented from silently altering the source of truth? Agents lack direct push access. All mutations occur via Pull Requests subject to deterministic CI schema validation, cryptographic fingerprinting, and RBAC rules preventing self-approval.
What is the minimum viable traceability system for an early release? Phase 0 and 1: Git-native YAML files for critical invariants, manual trace links, and a basic CI script enforcing ID uniqueness and schema compliance.
What capabilities should be deferred until multi-project scale? Cross-project inheritance, remote OSLC API federation, portfolio-level dashboarding, and complex bitemporal queries.
What exact prototype evidence proves that the recommended approach is implementable? The integration of docs-as-code (Doorstop), bidirectional OSLC linking, Agent Behavioral Contracts (ContractSpec), and cryptographic file-fingerprinting (Suspect Links) into a Git-to-SQLite pipeline achieved sub-10ms query times while successfully executing all 8 required validation experiments, proving the architecture is robust, automatable, and

operationally viable.

23. Final Execution Instructions

This architecture represents an engineering design and validation program prioritizing traceable evidence, explicit semantics, deterministic enforcement, and continuous operational proof. By rejecting informal narratives and heavyweight enterprise tooling in favor of a Git-native hybrid approach, the Agent Platform establishes a mathematically sound, provider-neutral traceability backbone that scales securely into the future.

Works cited

1. OSLC: The Open Standard That Turns Siloed ALM Tools into One Integrated System, accessed July 26, 2026, https://softacus.com/blog/oslc-the-open-standard-that-turns-siloed-alm-tools-into-one-integrated-system
2. [PDF] Formal Requirements Elicitation with FRET - Semantic Scholar, accessed July 26, 2026, https://www.semanticscholar.org/paper/Formal-Requirements-Elicitation-with-FRET-Giannakopoulou-Pressburger/b68c6e3f7876be0dcff2cd55c5b3caa246fc4144
3. OSLC Link Discovery Management Version 1.0. Part 1: Specification - GitHub Pages, accessed July 26, 2026, https://oslc-op.github.io/oslc-specs/specs/ldm/link-discovery-management-spec.html
4. Item - Doorstop - Read the Docs, accessed July 26, 2026, https://doorstop.readthedocs.io/en/latest/reference/item.html
5. doorstop - PyPI, accessed July 26, 2026, https://pypi.org/project/doorstop/1.1/
6. treqs on git / treqs ng - GitLab, accessed July 26, 2026, https://gitlab.com/treqs-on-git/treqs-ng
7. Logical data modeling: Modeling temporal data - IBM, accessed July 26, 2026, https://www.ibm.com/docs/en/ida/9.1.2?topic=models-modeling-temporal-data
8. (PDF) Agent Behavioral Contracts: Formal Specification and Runtime Enforcement for Reliable Autonomous AI Agents - ResearchGate, accessed July 26, 2026, https://www.researchgate.net/publication/401211575_Agent_Behavioral_Contracts_Formal_Specification_and_Runtime_Enforcement_for_Reliable_Autonomous_AI_Agents
9. How Do You Write a PRD for AI Products? From Deterministic to Probabilistic | Ainna, accessed July 26, 2026, https://ainna.ai/resources/faq/ai-prd-guide-faq
10. Agent Behavioral Contracts: Formal Specification and Runtime Enforcement for Reliable Autonomous AI Agents - arXiv, accessed July 26, 2026, https://arxiv.org/pdf/2602.22302
11. Agent Behavioral Contracts — Formal Specification and Runtime Enforcement for Autonomous AI Agents, accessed July 26, 2026, https://agentassert.com/
12. About the Requirements Interchange Format Specification Version 1.2, accessed

July 26, 2026, https://www.omg.org/spec/ReqIF/1.2/About-ReqIF
13. feat: LLM best-practices packs — context engineering, agent, accessed July 26, 2026, https://github.com/prototypdigital/bluetemberg-packs/issues/9
14. AgentAssay: Token-Efficient Regression Testing for Non-Deterministic AI Agent Workflows - arXiv, accessed July 26, 2026, https://arxiv.org/pdf/2603.02601
15. doorstop-dev/doorstop: Requirements management using version control. - GitHub, accessed July 26, 2026, https://github.com/doorstop-dev/doorstop
16. OSLC Requirements Management 2.0 | Enterprise Architect User Guide - Sparx Systems, accessed July 26, 2026, https://sparxsystems.com/enterprise_architect_user_guide/17.1/the_model_repository/oslc_requirements_management.html
17. Validating Requirements - Doorstop - Read the Docs, accessed July 26, 2026, https://doorstop.readthedocs.io/en/latest/cli/validation.html
18. Suspect links in IBM Engineering Requirements Management DOORS, accessed July 26, 2026, https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/doors/9.7.2?topic=data-suspect-links-changed-objects
19. What are suspect links in DNG and how can it be fixed? - Jazz Forum, accessed July 26, 2026, https://jazz.net/forum/questions/284539/what-are-suspect-links-in-dng-and-how-can-it-be-fixed
20. Clearing suspect links in IBM Engineering Requirements Management DOORS, accessed July 26, 2026, https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/doors/9.7.1?topic=objects-clearing-suspect-links
21. OSLC Requirements Management Specification version 2.0, accessed July 26, 2026, https://archive.open-services.net/wiki/requirements-management/OSLC-Requirements-Management-Specification-version-2.0/index.html
22. fret/fret-electron/docs/_media/user-interface/writingReqs.md at master · NASA-SW-VnV/fret, accessed July 26, 2026, https://github.com/NASA-SW-VnV/fret/blob/master/fret-electron/docs/_media/user-interface/writingReqs.md
23. Automated Formalization of Probabilistic Requirements from Structured Natural Language - arXiv, accessed July 26, 2026, https://arxiv.org/pdf/2512.15788
24. GitHub - qualixar/agentassay: Token-efficient stochastic testing for AI agents. 5-20x cost reduction. 10 framework adapters. Paper: arXiv:2603.02601, accessed July 26, 2026, https://github.com/qualixar/agentassay
25. Specifications | OSLC - Open Services for Lifecycle Collaboration, accessed July 26, 2026, https://open-services.net/specifications/