ETIS WHITE PAPER SERIES

WP-002

Repository-Centered Engineering

Why the Repository Is Becoming the Engineering System of Record

William T. O'Connell, Ph.D.

Adjunct Professor of Computer Science, Loyola University Chicago
Former IBM Distinguished Engineer and CTO, IBM Consulting Quality and Delivery
Creator, Engineering Trustworthy Intelligent Systems (ETIS)

July 2026

ETIS WP-002 | Version 1.0

Executive Abstract

Software repositories began as places to preserve source code. They are becoming something more consequential: the operational surface on which modern engineering organizations express intent, coordinate change, enforce policy, preserve decisions, verify claims, release software, and learn from production. In AI-assisted and increasingly agentic delivery environments, this shift accelerates. Humans and software agents both need authoritative context. Organizations need a durable record of what was requested, what changed, who or what produced it, how it was reviewed, what evidence justified release, and what happened after deployment.

Repository-centered engineering is the disciplined practice of treating the repository and its connected delivery environment as the engineering system of record. It does not mean that every organizational document belongs in Git, nor that a repository replaces product management, service management, enterprise architecture, or governance systems. It means that the evidence needed to understand, challenge, reproduce, release, operate, and improve a system remains connected to versioned artifacts and reviewable workflow. Requirements link to issues; issues bound change; branches isolate work; pull requests present engineering arguments; automated checks test claims; provenance records how artifacts were produced; releases establish controlled baselines; runbooks and postmortems return operational learning to the engineering record.

This paper argues that repository-centered engineering is no longer a developer preference. It is an emerging operating model for trustworthy software delivery. The direction is reinforced by GitHub's protected branches, rulesets, CODEOWNERS, pull-request reviews, and automation; DORA research on version control, continuous integration, small batches, fast feedback, and documentation; NIST's Secure Software Development Framework; SLSA provenance; CISA software bill of materials guidance; OpenSSF repository security practices; and current life-cycle and architecture standards. These sources do not prescribe one unified repository methodology. Together, however, they point toward a common conclusion: engineering trust increasingly depends on connected, enforceable, and inspectable evidence.

The central thesis is straightforward: code is only one part of the engineering record. The repository becomes professionally valuable when it preserves the chain from intent to operation - not as artifact accumulation, but as a coherent basis for judgment, accountability, automation, and controlled change.

Core thesis

The repository is becoming the place where an organization turns engineering intent into controlled change - and controlled change into defensible evidence.

Key propositions

- The repository is not merely a code container; it is a coordination, control, and evidence environment.
- A pull request is not merely a merge request; it is a compact engineering argument about why a change is safe enough to integrate.
- Repository structure and metadata increasingly serve both humans and engineering agents as executable context.
- Policy becomes credible when it is enforced through workflow, permissions, required checks, provenance, and review obligations.
- Evidence quality matters more than artifact volume. A strong record lets another competent reviewer reconstruct intent, decisions, validation, release, and operational learning.

- Repository-centered engineering does not replace standards or enterprise systems. It operationalizes their expectations at the point where engineering work is performed.

1. From Source Control to Engineering System of Record

The repository earned its place in software engineering by solving a narrow but essential problem: preserving source code and managing concurrent change. Distributed version control then made branching, merging, and collaboration inexpensive enough to become routine. Yet the most important evolution was not technical. The repository gradually became the shared surface where engineering work could be proposed, discussed, reviewed, automated, integrated, released, and audited.

Modern repositories now contain far more than implementation. They preserve requirements, architecture descriptions, decision records, test plans, infrastructure definitions, workflows, configuration, security policies, release notes, operational procedures, and evidence generated by delivery pipelines. Connected issue trackers, pull-request systems, artifact registries, deployment platforms, and observability tools extend that record beyond a single directory tree. The repository therefore functions less like a filing cabinet and more like an operational control plane for engineering change.

This direction is consistent with the wider life-cycle view of software engineering. ISO/IEC/IEEE 12207:2026 applies across conception, development, operation, support, and retirement. ISO/IEC/IEEE 42010:2022 treats architecture descriptions as sustained engineering assets rather than one-time diagrams. DORA identifies version control, continuous integration, trunk-based development, test automation, documentation, and small-batch work as capabilities associated with effective delivery. None of these sources declares that the repository must become the system of record. Their combined logic makes the direction difficult to avoid: the life cycle needs a durable place where intent, change, evidence, and learning remain connected. [1-4]

The phrase system of record should be used carefully. A repository does not replace customer systems, financial records, human-resources systems, service-management platforms, or regulatory archives. It is the engineering system of record: the authoritative, reviewable chain needed to understand how the system was conceived, changed, verified, released, and operated. Some records may remain in external systems, but material engineering claims should link to their authoritative evidence rather than depend on memory, email, or unstructured conversation.

The repository becomes authoritative not because it contains everything, but because it connects everything necessary to explain and defend engineering change.

2. The Engineering Record Is a Chain, Not a Folder Tree

Weak repository practice is often mistaken for an organizational problem: files are hard to find, folders are inconsistent, or documentation is incomplete. Those problems matter, but repository-centered engineering addresses something deeper. The central object is not the folder tree. It is the chain of reasoning and evidence that allows a reviewer to move from an engineering claim to its support.

Consider a seemingly simple claim: the new authentication change is ready for release. A professional reviewer should be able to trace that claim to the business or security intent that justified the work; the issue that bounded scope; the architecture or threat-model decision that shaped the solution; the branch and commits that implemented it; the pull request that exposed the change; the reviewers responsible for affected components; the test and scan results that challenged the implementation; the deployment and rollback plan; and the release record that established the approved baseline. After deployment, telemetry, incidents, and post-release findings should return to the same engineering history.

This chain changes the meaning of documentation. Documentation is no longer a parallel narrative written after implementation. It becomes part of the change itself. An architecture decision record can explain why a dependency was introduced. A threat model can establish what attacks were considered. A test result can support an acceptance criterion. A runbook can prove that the system is operable. A postmortem can show that operational learning changed architecture, tests, or release controls. The value lies in connection and use, not in document count.

A repository can therefore be well organized and still be weak. It may contain polished documents that are stale, duplicated, detached from code, or impossible to connect to releases. Conversely, a compact repository can be strong if its important claims, decisions, changes, and evidence are current, linked, and reviewable. The goal is not maximal documentation. It is minimal ambiguity at the points where professional judgment is required.

The repository-centered evidence chain

Lifecycle concern	Repository expression	Question answered	Typical evidence
Intent	Requirement, issue, acceptance criteria	Why should this change exist?	Stakeholder outcome, constraints, examples
Design	Architecture view, ADR, threat model	Why is this approach suitable?	Boundaries, alternatives, risks, authority
Change	Branch, commits, pull request	What changed and why?	Diff, linked issue, rationale, AI disclosure
Verification	CI/CD, tests, scans, review	What evidence challenges the claim?	Results, failures, reviewer findings, provenance
Release	Tag, release record, deployment definition	What exactly was approved?	Immutable baseline, SBOM, approvals, rollback
Operation	Runbook, telemetry links, incidents	Did the system behave as expected?	SLOs, alerts, overrides, incident record
Learning	Postmortem, backlog, revised controls	What changed because of experience?	Corrective actions, updated tests and decisions

3. Pull Requests as Engineering Arguments

The pull request is the most revealing object in repository-centered engineering because it sits at the boundary between individual production and shared acceptance. In its weakest form, it is a technical mechanism for asking permission to merge. In its strongest form, it is a concise engineering argument: this change is necessary, bounded, understandable, sufficiently tested, consistent with the architecture, reviewed by the right people, and safe enough to integrate under known limitations.

GitHub's review model, CODEOWNERS, protected branches, and rulesets demonstrate how review obligations can become enforceable workflow rather than convention. Protected branches can require pull requests, approving reviews, passing status checks, signed commits, or linear history. CODEOWNERS can route changes to responsible reviewers, and branch rules can require their approval. These capabilities matter because accountability cannot depend entirely on good intentions. Engineering discipline becomes more reliable when the repository prevents unreviewed or unverified change from silently crossing important boundaries. [5-9]

A strong pull request should tell a reviewer what problem is being solved, what is deliberately out of scope, what assumptions shaped the implementation, how the change was tested, what risks remain, whether AI generated or materially influenced the work, and what recovery path exists if the change fails. That information should not become a burdensome essay. The discipline is to make the engineering argument proportional to consequence. A typo needs little explanation; an authorization change, model-routing update, schema migration, or production agent permission requires substantially more.

AI-assisted delivery raises the value of this argument. Agents can create large, plausible diffs quickly. Human reviewers may be tempted to inspect only whether automated checks passed. Yet the core review question is not whether the syntax is acceptable. It is whether the change fits the system, satisfies the real intent, preserves safety and operability, and introduces risks that the automated checks do not understand. As artifact production accelerates, the pull request becomes a scarce-attention interface: the place where generated work must be converted into humanly defensible change.

A diff shows what changed. A professional pull request explains why the change deserves to become part of the system.

4. The Repository as Executable Context for Humans and Agents

Repository quality has always affected human productivity. It now affects machine performance directly. Coding agents, review agents, modernization tools, and autonomous delivery workflows derive their understanding from the repository: its structure, naming, architecture records, tests, build instructions, policies, examples, and local conventions. In that sense, repository content becomes executable context. It does not merely describe the system; it shapes the behavior of the agents allowed to change it.

This introduces a new form of technical debt: context debt. A repository with missing architecture, inconsistent conventions, stale instructions, weak tests, or undocumented operational constraints produces unreliable human decisions and unreliable agent output. The agent may generate code that compiles while violating hidden assumptions, duplicating existing capabilities, bypassing controls, or creating an operational burden. The failure is not only in the model. The engineering environment failed to provide authoritative context and enforce meaningful boundaries.

Repository-centered engineering therefore treats context as a controlled asset. Project instructions should identify authoritative sources, coding and security expectations, dependency rules, test commands, architectural constraints, prohibited changes, escalation points, and evidence obligations. Those instructions should be versioned, reviewed, and tested through use. They should not attempt to encode every organizational fact. Their purpose is to reduce consequential ambiguity and direct humans and agents toward the right evidence.

The repository also becomes a boundary for authority. Agent access should be scoped to the minimum repositories, branches, tools, secrets, and environments needed for the task. Proposed changes should remain isolated until reviewed. High-consequence areas should require responsible owners. Automated checks should challenge generated work. Every agent action should leave provenance sufficient to reconstruct what was requested, what context was used, what tools were invoked, and what outputs were accepted. This is where repository design, identity, security, and governance converge.

5. Policy as Workflow, Governance as Architecture

Organizations often describe governance as a set of policies, review boards, and approval documents. Those mechanisms have value, but they are weakest when detached from engineering execution. A policy that says

security review is required but allows direct pushes to production branches is not an effective control. A standard that requires traceability but leaves requirements, tests, and releases disconnected is not operationalized. A responsible-AI principle that requires human oversight but does not define approval points, permissions, telemetry, or intervention paths remains aspirational.

Repository-centered engineering turns governance into workflow. Branch protections define how change may enter controlled baselines. Required reviewers establish decision rights. CODEOWNERS map component authority. CI/CD workflows implement tests, scans, policy checks, and evidence generation. Release rules define what constitutes an approved baseline. Signed commits, attestations, and provenance improve confidence in origin. Issue templates and pull-request templates can require risk, AI use, testing, and rollback information. The repository becomes the place where architecture and policy constrain change before consequences reach production.

This does not eliminate human governance. It improves the quality of human governance by presenting decision-makers with connected evidence and by making routine obligations enforceable. Humans still determine which controls are appropriate, when exceptions are justified, and who accepts residual risk. The key distinction is that governance is no longer applied at the edge of delivery as a final inspection. It is embedded into the architecture and workflow of change.

NIST's SSDF is intentionally implementation-neutral, but it emphasizes protecting development environments, producing well-secured software, responding to vulnerabilities, maintaining provenance, and integrating secure practices into the chosen life cycle. NIST's DevSecOps reference work explicitly addresses the gap between high-level SSDF outcomes and practical implementation. OpenSSF's source-code-management best practices, Scorecard, and OSPS Baseline similarly turn repository configuration into visible security posture. These sources reinforce a practical conclusion: repository controls are becoming part of the security and governance architecture of the software system. [10-14]

Three layers of repository-centered governance

Layer	Primary concern	Repository controls	Evidence
Design time	What may be built and under what authority?	Requirements, ADRs, ownership, threat model, policy files	Intent, boundaries, accepted risk
Delivery time	Why is this change safe enough to integrate and release?	Protected branches, reviews, checks, attestations, release gates	Review record, test results, provenance, approvals
Runtime feedback	Is the deployed system behaving within expectations?	Deployment definitions, runbooks, telemetry links, incident workflow	Operational outcomes, overrides, rollback, postmortem

6. Provenance, Supply-Chain Integrity, and the Release Baseline

A repository can preserve a trustworthy history of source change while the delivered artifact remains uncertain. Modern software is assembled from source, dependencies, build tools, containers, generated assets, infrastructure definitions, and external services. The security question therefore extends beyond who committed the code. It includes what inputs were used, which process produced the artifact, whether the build environment was controlled, whether the artifact was altered, and whether the deployed release corresponds to the approved source.

SLSA formalizes provenance as verifiable information about where, when, and how an artifact was produced. Its guidelines are designed to make software supply chains more resistant to tampering and to help consumers decide whether to trust a package. NIST SSDF calls for protecting release integrity and preserving provenance. CISA describes the software bill of materials as a formal inventory of software components and their supply-chain relationships. OpenSSF Scorecard evaluates repository practices such as branch protection, dependency pinning, review, and automated updates. These mechanisms address different parts of the problem, but together they establish a broader release principle: an approved release is not merely a commit hash. It is a traceable, reproducible, and inspectable baseline. [10, 15-20]

Repository-centered engineering connects that baseline to the engineering record. The release should identify the approved change set, test and scan evidence, dependency inventory, build provenance, configuration, known limitations, rollback path, and accountable approvers. For intelligent systems, the release unit may also include prompts, policies, model and routing configuration, retrieval indexes, tool permissions, evaluation data, and monitoring rules. Every independently changeable element that can alter system behavior should have ownership, versioning, validation, and rollback thinking.

This discipline is not only for regulated or safety-critical software. Supply-chain compromise, dependency failure, configuration drift, and unreviewed automation can affect any organization. The cost of reconstructing a release after an incident is highest when the engineering record is fragmented. Repository-centered release evidence reduces that uncertainty before the incident occurs.

A release is not just what the team intended to ship. It is the specific, verifiable system configuration that the organization accepted responsibility for operating.

7. Small Batches, Fast Feedback, and the Economics of Review

Repository-centered engineering is sometimes criticized as process overhead. Poorly designed workflow can indeed become bureaucracy: large templates, ceremonial approvals, duplicated evidence, and gates that do not improve decisions. The answer is not to remove discipline. It is to design the repository around small batches, fast feedback, and risk-proportionate evidence.

DORA's research consistently emphasizes version control, continuous integration, trunk-based development, automated testing, documentation, and small-batch work. Small batches make changes easier to understand, test, review, recover, and learn from. Trunk-based development limits long-lived divergence. Continuous integration reveals conflicts and failures early. Fast code review improves delivery performance. High-quality documentation amplifies other technical capabilities. These findings explain why repository workflow can improve speed and safety simultaneously: it shortens the distance between an engineering decision and the evidence that challenges it. [2-4, 21]

AI strengthens the economic case for small batches. Agents can produce more change than reviewers can absorb. Large generated diffs create an illusion of productivity while shifting cost into review, integration, debugging, and operational risk. The strongest operating model does not ask agents to generate as much as possible. It asks them to produce bounded, testable, reversible changes that fit available human and automated verification capacity.

The correct metric is not repository activity. Commit count, pull-request volume, lines changed, issue closure, and documentation pages can all increase while system quality declines. Repository-centered engineering should be evaluated by outcomes: lead time, change failure, recovery, escaped defects, review effectiveness, traceability, release confidence, security posture, operational reliability, and the quality of learning. The

repository is valuable because it makes those outcomes explainable, not because it creates more visible motion.

8. Failure Modes of Repository-Centered Engineering

Any engineering model can be implemented badly. Repository-centered engineering fails when the repository becomes an artifact warehouse, a surveillance system, a substitute for communication, or a rigid process imposed without regard to risk and team context.

The first failure mode is documentation theater. Teams create templates, checklists, and records to satisfy a gate, but the artifacts are not used to make decisions. They become stale immediately and reduce trust in the repository. The remedy is to require only evidence that supports a real claim, review, release, or operational need - and to update it as part of the work rather than after the deadline.

The second failure mode is broken traceability. Requirements, issues, pull requests, tests, releases, and incidents exist, but they are not connected. Reviewers must search manually or ask the team where evidence lives. The remedy is not universal tooling; it is consistent identifiers, links, naming, ownership, and repository navigation that make the engineering chain visible.

The third failure mode is automated compliance without judgment. Passing checks can create false confidence. Tests may cover the wrong behavior, scans may miss architectural risk, required reviewers may approve superficially, and provenance may prove how a flawed artifact was built. Automation should reduce routine uncertainty, not replace professional skepticism.

The fourth failure mode is process disproportion. Low-risk changes receive the same burden as high-consequence changes, encouraging teams to bypass or game the workflow. Repository controls should scale with impact, authority, reversibility, data sensitivity, and operational consequence.

The fifth failure mode is repository exceptionalism. Teams assume that if information is not in Git it does not matter. In reality, customer research, legal obligations, financial controls, production telemetry, service-management records, and enterprise architecture may live elsewhere. Repository-centered engineering succeeds when it connects authoritative systems, not when it pretends to replace them.

The final failure mode is confusing transparency with punishment. A visible engineering record should support learning and accountability, not incentivize concealment. If teams are punished for surfacing uncertainty, failed experiments, rejected AI suggestions, or operational mistakes, the repository will become polished fiction. Trust requires an environment in which evidence can reveal weakness before weakness becomes harm.

Repository anti-patterns and corrective moves

Anti-pattern	What it looks like	Corrective move
Artifact warehouse	Many files, little connection to decisions or releases	Retain only decision-useful evidence and link it to workflow
Merge-button pull request	Diff with no rationale, risk, or validation story	Require a proportionate engineering argument
Green-check confidence	Passing automation treated as proof of fitness	Combine checks with architecture, risk, and operational review
One-size-fits-all governance	Same burden for typo and authorization redesign	Scale controls to consequence and reversibility
Invisible agent work	AI contribution cannot be reconstructed	Preserve task, context, tools, changes,

Anti-pattern	What it looks like	Corrective move
		review, and acceptance evidence
Repository isolation	Ignores external systems of authority	Link authoritative records and define boundaries

9. An Operating Model for Repository-Centered Engineering

Organizations do not become repository-centered by adopting a directory template. The model requires aligned practices, platforms, decision rights, and incentives. The implementation should begin with the engineering claims that matter most: what the system is intended to do, who may change it, what evidence is required for integration and release, how production behavior is observed, and how learning returns to the system.

A practical operating model has five elements. First, define the repository's authority boundary. Identify which artifacts are authoritative in the repository, which external systems remain authoritative, and how links are preserved. Second, create a navigable evidence architecture. A reviewer should be able to locate current requirements, architecture, decisions, plans, tests, release records, operational guidance, and known risks without asking the original author. Third, implement workflow controls. Protected branches, ownership, required checks, secrets management, dependency policy, provenance, and release gates should match the consequence of the system. Fourth, establish professional review. Pull requests and phase gates should challenge engineering claims, not merely confirm completion. Fifth, close the operational loop. Runbooks, telemetry, incidents, postmortems, and maturity improvements must feed back into requirements, architecture, testing, and governance.

The rollout should be incremental. Teams can begin by standardizing a small set of high-value practices: issue-linked branches, meaningful pull-request descriptions, required review for protected areas, automated tests and scans, release records, and a visible risk and decision history. Once those practices are reliable, the organization can add provenance, SBOMs, policy as code, agent-specific controls, evidence dashboards, and cross-repository governance.

Leadership behavior is decisive. If leaders reward feature volume while treating review, testing, documentation, and operational readiness as delay, teams will optimize around the repository controls rather than through them. If leaders use the engineering record to make decisions, allocate risk, learn from failure, and recognize professional judgment, the repository becomes living organizational capability.

A maturity path

Stage	Repository role	Characteristic practices	Primary risk
1. Code store	Preserves implementation	Basic version control, informal review	Knowledge remains in people and chat
2. Team workspace	Coordinates change	Issues, branches, PRs, CI, templates	Evidence exists but is inconsistent
3. Engineering record	Connects intent, decisions, verification, release	ADRs, traceability, required reviews, release evidence	Controls may remain project-specific
4. Governance surface	Enforces organizational policy	Rulesets, provenance, SBOM, policy checks, ownership	Automation may outpace judgment
5. Human-agent control plane	Provides context and bounded	Agent identity, scoped tools,	Diffused responsibility and

Stage	Repository role	Characteristic practices	Primary risk
	authority to agents	task provenance, behavioral evaluation	runtime consequence

10. The ETIS Perspective

Engineering Trustworthy Intelligent Systems (ETIS) treats repository-centered engineering as connective tissue across the life cycle. The repository is where intent, architecture, implementation, AI use, review, verification, release, operation, governance, and learning remain visible to one another. ETIS does not claim that Git or GitHub is sufficient for trustworthy engineering. Its claim is that trustworthy engineering becomes much harder when material decisions and evidence are fragmented, unversioned, or disconnected from the system they govern.

Several ETIS doctrines follow directly. Everything important leaves evidence. The model is not the system. AI proposes; engineers verify. Governance is architecture. Context is control. A demo is not operational proof. These are not slogans attached to a repository template. They define how the engineering record should function. An issue captures bounded intent. An architecture decision constrains implementation. A pull request presents a change for challenge. Tests and evaluations provide evidence. A release record establishes an accountable baseline. Observability and postmortems show whether the claims survived operational reality.

Repository-centered engineering is especially important for intelligent and agentic systems because behavior depends on more than source code. Prompts, context instructions, model configurations, policies, retrieval settings, identities, tool permissions, evaluation data, and monitoring rules may all change system behavior. ETIS therefore broadens the release and evidence model. These elements should be versioned, owned, validated, and connected to rollback and operational control.

The distinct contribution of ETIS is not that it replaces NIST, ISO, DORA, SLSA, OWASP, GitHub, DevSecOps, or SRE. It operationalizes their concerns within one repository-centered engineering lifecycle. NIST defines secure outcomes but not a project-specific evidence architecture. ISO defines life-cycle and architecture expectations but not a pull-request workflow. SLSA defines provenance but not product intent. DORA explains delivery capabilities but not agent authority. GitHub provides controls but not an engineering doctrine. ETIS supplies the connective model that helps teams decide what evidence to create, where it belongs, how it is reviewed, and how it supports trust.

The repository is not the system. It is the place where the organization preserves enough truth about the system to change it responsibly.

11. Implications for the Next Decade

The repository will continue to evolve from a developer tool into an enterprise engineering platform. Several directions are already visible.

First, repository instructions will become more formal and machine-consumable. Requirements, architecture constraints, security rules, test expectations, and operational procedures will increasingly be consumed by agents as task context. Organizations will invest in context quality because poor context creates poor autonomous work.

Second, identity and authority will become repository-native concerns. Engineering agents will have distinct identities, roles, permissions, approved tools, and evidence obligations. The question will no longer be only who approved the pull request, but which human and machine actors participated, what authority they exercised, and what controls constrained them.

Third, evidence will be generated continuously. Tests, scans, provenance, SBOMs, policy decisions, evaluation results, and operational signals will attach automatically to changes and releases. Reviewers will need better ways to distinguish meaningful evidence from automated noise.

Fourth, repositories will become cross-life-cycle knowledge graphs. Links among requirements, components, decisions, code, tests, releases, incidents, and owners will support impact analysis for both humans and agents. The competitive advantage will not be the volume of stored information; it will be the quality of connected, current, authoritative context.

Fifth, governance and platform engineering will converge. Repository rules, developer platforms, AI control planes, deployment systems, and observability will increasingly enforce a common operating model. Organizations that treat these capabilities as separate tools will struggle with inconsistency. Organizations that design them as one engineering system will gain speed, safety, and learning.

Finally, professional competence will be judged by the ability to create and challenge engineering arguments. As AI generates more code, designs, tests, and documentation, engineers will spend less time proving that an artifact exists and more time proving that it fits the system, satisfies the intent, respects authority, survives failure, and can be operated responsibly. The repository will be the principal arena in which that proof is assembled.

Conclusion

Software engineering is moving beyond source control without leaving it behind. The repository remains the place where code is preserved, but its professional role is expanding. It is becoming the shared engineering environment in which organizations define work, constrain change, review decisions, automate verification, protect release integrity, preserve provenance, and return operational learning to the system.

Repository-centered engineering should not be reduced to Git hygiene, documentation standards, or tool configuration. It is an operating model for controlled change. Its quality is measured by whether another competent reviewer can reconstruct what was intended, understand why a decision was made, see what changed, examine the evidence, identify accepted risk, reproduce the release, and learn what happened in operation.

This model becomes more important as AI and agents participate in delivery. Artifact generation becomes abundant; trustworthy context, architecture, review, authority, evidence, and accountability remain scarce. The organizations that benefit most from AI will not be those that produce the most changes. They will be those that can convert human and machine production into small, reviewable, verifiable, reversible, and operationally responsible change.

The repository is not the system, and it is not a substitute for engineering judgment. It is the place where judgment becomes visible, challengeable, enforceable, and durable. That is why it is becoming the engineering system of record.

Appendix A. Professional Alignment Crosswalk

Engineering concern	External alignment	Repository-centered implementation	Boundary
Life-cycle continuity	ISO/IEC/IEEE 12207:2026	Connects conception, development, operation, support, and retirement evidence	The standard does not prescribe a repository workflow
Architecture knowledge	ISO/IEC/IEEE 42010:2022	Architecture descriptions, viewpoints, ADRs, and ownership near implementation	The standard does not mandate Git or PRs
Delivery performance	DORA capabilities	Small batches, CI, trunk-based development, fast review, documentation	DORA does not define an evidence governance model
Secure development	NIST SSDF	Protected environments, controlled change, secure defaults, provenance, vulnerability response	SSDF is outcome-oriented and implementation-neutral
Supply-chain integrity	SLSA; CISA SBOM	Build provenance, dependency inventory, release verification	Provenance does not prove product fitness
Repository security	OpenSSF Scorecard; SCM best practices; OSPS Baseline	Branch protection, review, dependency policy, automation, security posture	Automated scoring cannot replace context-specific judgment
Workflow control	GitHub protected branches, rulesets, CODEOWNERS	Required reviews, status checks, signed commits, responsible ownership	Platform features need an engineering operating model
AI-assisted delivery	Agentic coding and review platforms	Repository instructions, scoped authority, task provenance, human approval	Tool capability does not transfer professional accountability

References

[1] ISO, "ISO/IEC/IEEE 12207:2026 - Software life cycle processes," 2026. Available online

[2] DORA, "Capabilities: Catalog." Available online

[3] DORA, "Trunk-based development." Available online

[4] DORA, "Working in small batches," updated Dec. 8, 2025. Available online

[5] GitHub, "About pull request reviews." Available online

[6] GitHub, "About code owners." Available online

[7] GitHub, "About protected branches." Available online

[8] GitHub, "Managing a branch protection rule." Available online

[9] GitHub, "About rulesets." Available online

[10] NIST, "Secure Software Development Framework (SSDF) Version 1.1," SP 800-218, 2022. Available online

[11] NIST NCCoE, "DevSecOps Practices and Secure Software Development." Available online

[12] OpenSSF, "Source Code Management Platform Configuration Best Practices." Available online

[13] OpenSSF, "OpenSSF Scorecard." Available online

[14] OpenSSF, "Open Source Project Security Baseline," 2025. Available online

[15] SLSA, "About SLSA," Specification v1.2. Available online

[16] SLSA, "Provenance." Available online

[17] CISA, "Software Bill of Materials (SBOM)." Available online

[18] CISA, "2025 Minimum Elements for a Software Bill of Materials," Aug. 22, 2025. Available online

[19] OpenSSF, "Reducing Security Risks in Open Source Software at Scale," Jan. 19, 2022. Available online

[20] OpenSSF, "How Intel Uses OpenSSF Scorecard," Mar. 25, 2024. Available online

[21] DORA, "How to enable software delivery teams to innovate with generative AI," Aug. 16, 2024. Available online

[22] ISO, "ISO/IEC/IEEE 42010:2022 - Architecture description," 2022. Available online

[23] NIST, "Secure Software Development Practices for Generative AI and Dual-Use Foundation Models," SP 800-218A, 2024. Available online

[24] DORA, "Continuous integration." Available online

[25] DORA, "Continuous delivery." Available online

About the Author

William T. O'Connell, Ph.D. is an Adjunct Professor of Computer Science at Loyola University Chicago and the creator of the Engineering Trustworthy Intelligent Systems (ETIS) framework. His professional career spans more than four decades of software engineering, architecture, delivery, quality, and technology leadership, including service as an IBM Distinguished Engineer and CTO for IBM Consulting Quality and Delivery. His current work focuses on repository-centered engineering, engineering evidence, trustworthy intelligent systems, agentic software delivery, and the preparation of students and practicing engineers for increasingly AI-assisted engineering environments.

This white paper is part of the ETIS White Paper Series, which examines the engineering practices required to build, govern, review, operate, and sustain trustworthy software and intelligent systems.

Suggested citation: O'Connell, W. T. (2026). Repository-Centered Engineering: Why the Repository is Becoming the Engineering System of Record. ETIS White Paper Series, WP-002, Version 1.0.