Evidence-Gated Agentic Development

Applying the History of Distributed Systems to AI-Operated Software Delivery

Author: Kris Christopher / KrisCodes2
Case study: Conexus
Date: July 2026
Status: Technical white paper based on a pre-release internal platform

---

Abstract

AI coding systems are increasing the rate at which organizations can produce software changes, but code-generation speed does not solve the problems of coordination, verification, authority, provenance, or recovery. When multiple AI agents operate across shared repositories, machines, pipelines, and business systems, they exhibit the defining characteristics of a distributed system: independent execution, message-based communication, partial state, asynchronous timing, concurrent work, and partial failure.

This white paper develops **Evidence-Gated Agentic Development**, a software-delivery model in which AI agents may retrieve context, plan bounded work, modify software, generate tests, and repair eligible failures, while commit, push, promotion, deployment, and exception authority remain constrained by deterministic verification, signed evidence, policy gates, and human-controlled escalation.

The model draws on the historical development of packet-switched networking, logical clocks, Byzantine fault analysis, distributed snapshots, remote procedure calls, consensus, distributed locking, web-scale data systems, cluster orchestration, observability, continuous integration, and trunk-based development. Conexus is presented as a pre-release case study that combines an AI layer, a verification engine, and a shared Brain and command center to implement this control philosophy on customer-controlled infrastructure.

The principal conclusion is that AI agents should be treated as untrusted distributed workers rather than autonomous authorities. The central engineering question is not whether an agent can generate a plausible change, but whether the surrounding system can establish that the change is authorized, correctly evaluated, traceable, and eligible to advance.

---

1. Introduction

Software engineering is moving from human-paced production toward machine-paced production. A single developer can now coordinate several coding agents, research agents, test-generation agents, and repair loops. The marginal cost of producing another implementation has fallen sharply. The relative cost of reviewing, integrating, verifying, and governing those implementations has increased.

This shift creates a control problem.

A conversational agent can report that work is complete, but its report does not prove that the correct tests ran, the security tool was present, the evidence belongs to the current commit, the requirements were interpreted correctly, or the agent remained within its authorized scope. Multiple agents can also collide, repeat earlier work, operate from stale context, or continue acting after ownership changes.

These are not uniquely artificial-intelligence problems. They are distributed-systems problems expressed through a new class of worker.

Conexus applies this historical lens through three connected capabilities:

1. **AI layer:** reusable skills, workflows, retrieval conventions, and model interfaces.
2. **Verification layer:** a broad GitLab CI/CD verification engine that produces explicit operational outcomes and evidence.
3. **Brain and command center:** durable knowledge, coordination state, work ownership, messages, checkpoints, evidence references, operational visibility, and human approval.

The platform's central value proposition is controlled leverage: use machine-speed development without granting machine output unbounded authority.

---

2. Distributed-systems foundations

2.1 Independent actors and partial failure

A distributed system contains independent processes that coordinate through communication. No process has perfect instantaneous knowledge of the others. A component may fail while the rest of the system remains alive. Silence may indicate a crash, network delay, overload, lost response, or slow execution.

Agent swarms have the same ambiguity. A stopped response does not establish whether a task failed, completed, partially modified files, or produced a commit. Liveness and completion must therefore be recorded separately.

2.2 Protocols and boundary verification

The early ARPANET connected heterogeneous computers through explicit host protocols. RFC 1, published in April 1969, discussed host software, messages, links, connection primitives, congestion behavior, and interface error checking.

The enduring design principle is that heterogeneous participants require explicit contracts. A prompt is not a durable protocol. Agent coordination requires defined identities, state transitions, message formats, ownership rules, error states, and evidence contracts.

2.3 Logical time and causal identity

Lamport's 1978 work on logical clocks formalized the happened-before relationship. Distributed events may be causally related even when physical timestamps are unreliable, and some events are concurrent rather than globally ordered.

For agentic software delivery, causal identity should bind:

- Goal and requirement
- Work item
- Agent session
- File changes
- Commit
- Pipeline
- Policy version
- Toolchain
- Evidence bundle
- Approval
- Deployment
- Operational outcome

A "latest green" indicator is insufficient when the evidence may refer to a superseded commit.

2.4 Byzantine and arbitrary faults

The Byzantine Generals Problem formalized agreement in the presence of participants that may produce inconsistent information. AI agents are not normally malicious participants, but they can produce arbitrary or confidently incorrect outputs.

The appropriate response is not to demand that the generator become infallible. It is to separate generation from verification and authorization.

2.5 Consistent snapshots and recoverable state

The Chandy-Lamport snapshot algorithm showed how to record a consistent global state without stopping a distributed system. Agent workflows similarly need structured checkpoints that survive the loss of a live session.

A recoverable checkpoint should preserve completed work, pending work, decisions, modified files, blockers, evidence status, and the next bounded action.

2.6 Remote calls and hidden failure modes

Remote procedure call and network-file-system abstractions made remote resources easier to use. They also demonstrated that transparent interfaces can hide latency, serialization, retry, duplication, authorization, and partial-failure semantics.

Agent tool calls, pipeline dispatch, shared-state updates, and deployment requests require explicit handling of the same concerns.

2.7 Consensus, locks, leases, and fencing

Consensus algorithms such as Paxos and Raft address replicated agreement. Distributed lock services such as Chubby provide coarse-grained coordination.

Agentic systems do not require general consensus for every edit, but they require machine-verifiable ownership. Claims provide intent visibility. Leases provide time-bounded authority. Fencing tokens allow a protected resource to reject a stale worker after reassignment.

---

3. The web-scale transition

The early 2000s normalized the design assumption that hardware and network components fail continuously at scale.

Google's File System used commodity hardware while providing fault tolerance and high aggregate performance. MapReduce hid partitioning, scheduling, failure handling, and communication behind a constrained programming model. Bigtable provided structured storage across large clusters. Chubby supplied reliable low-volume coordination. Amazon Dynamo combined consistent hashing, versioning, quorum techniques, anti-entropy, and conflict reconciliation to prioritize high availability for selected workloads.

These systems demonstrated three principles relevant to agentic development:

1. Failure must be treated as an expected operating condition.
2. General complexity can be controlled through constrained interfaces.
3. Tradeoffs must be explicit under failure.

Conexus applies different failure policies to different risks. Verification fails closed when required tools or evidence are missing. Work leases may expire and be reassigned. Human judgment remains authoritative for decisions that are not reducible to deterministic checks.

---

4. Orchestration and observability

Cluster managers such as Borg and Kubernetes use declarative desired state and continuous reconciliation. The controller does not merely issue a command and assume success. It compares observed state with desired state and attempts to close the difference.

Evidence-gated development uses the same pattern:

- A PRD defines desired outcomes.
- Work items define bounded obligations.
- Verification profiles define required checks.
- Evidence schemas define required proof.
- Policy determines whether observed results permit advancement.

Distributed tracing systems such as Dapper demonstrate the need to reconstruct one operation across many services. Agentic delivery requires a wider trace: goal, requirement, agent, commit, pipeline, evidence, approval, deployment, and outcome.

Observability therefore includes not only infrastructure metrics, logs, and traces, but also decision provenance and evidence provenance.

---

5. Continuous integration and mainline convergence

A software organization is itself a distributed system. Developers hold partial knowledge, work concurrently, and converge through version control, code review, testing, and release processes.

Long-lived feature branches delay convergence and create expensive integration events. Continuous integration shifts the authority question from whether one branch works to whether the combined system remains valid. Trunk-based development supports this through small batches and frequent mainline integration.

AI agents increase the production rate and therefore intensify the integration problem. When candidate changes become abundant, commit and promotion authority must be derived from verifiable state rather than conversational confidence.

A mainline-oriented implementation of the model can be termed **Evidence-Gated Agentic Trunk Development**.

---

6. Definition of Evidence-Gated Agentic Development

Evidence-Gated Agentic Development is a software-delivery model in which AI agents may retrieve, plan, generate, test, and repair bounded changes, while consequential authority remains constrained by deterministic verification, signed evidence, policy evaluation, and human-controlled exceptions.

The model has six defining properties.

6.1 Agents are producers, not final authorities

An agent may generate a candidate and explain its reasoning. Independent controls determine whether the candidate satisfies the required conditions.

6.2 Work is bounded and attributable

Every consequential action is connected to an approved requirement, scoped work item, session identity, and ownership record.

6.3 Verification is fail-closed

A missing tool, invalid artifact, infrastructure failure, or unscheduled required job cannot be silently converted into success.

6.4 Evidence is bound to the evaluated state

Results are tied to the exact commit, pipeline, policy, toolchain, artifacts, and approvals that produced them.

6.5 Repair cannot weaken the gate

Automated repair may satisfy a requirement but may not lower the requirement, delete meaningful tests, disable scanners, or broaden exclusions without separately authorized policy change.

6.6 Humans retain exception and policy authority

Routine operations may become machine-executed. Humans remain accountable for architecture, risk tolerance, policy, requirements, sensitive exceptions, and consequential approvals.

---

7. The gated operating loop

The model can be implemented as a nine-stage loop.

7.1 Observe

Collect failures, active sessions, open work, resource state, security findings, deployment state, and business priority.

7.2 Recall

Retrieve authoritative symbols, documentation, architecture decisions, prior failures, successful repairs, applicable requirements, and existing ownership.

7.3 Plan

Decompose approved goals into small work items with dependencies, acceptance criteria, file expectations, tests, security controls, and evidence requirements.

7.4 Claim

Register the session, claim the work, obtain leases, declare file intent, and check for conflicts.

7.5 Execute

Generate code, tests, documentation, configuration, and supporting artifacts within the authorized scope.

7.6 Verify

Run deterministic tests, quality analysis, security controls, supply-chain checks, documentation gates, and evidence validation.

7.7 Approve

Apply policy and pause for human approval when the action exceeds delegated authority.

7.8 Ship and observe

Promote the approved change and measure technical and business outcomes.

7.9 Learn

Preserve the result, evidence, failure signature, repair, decision, and operational outcome for future retrieval.

---

8. Conexus case study

8.1 AI layer

The Conexus AI layer supplies reusable skills, hooks, workflows, agent interfaces, and retrieval-first conventions. It improves consistency and reduces repeated discovery. It does not own CI policy or final release authority.

8.2 Verification layer

The documented Conexus verification design contains more than 69 GitLab jobs spanning testing, coverage, linting, type analysis, static and dynamic security, secrets, dependency risk, licenses, software bills of materials, image analysis, documentation, signing, evidence, and release controls.

The remediation contract distinguishes PASS, FAIL, SKIPPED-NOT-APPLICABLE, INFRASTRUCTURE-ERROR, TOOL-MISSING, and EVIDENCE-INVALID. A job may not report success merely because an artifact exists; the intended tool must execute, the output must be structurally and semantically valid, and applicable policy must allow the result.

8.3 Brain and command center

The Brain stores durable project knowledge and operational state. Its coordination functions include session registration, heartbeats, file claims, collision visibility, work leases, fencing tokens, durable messages, receipts, checkpoints, handoffs, and evidence references.

A Fastify gateway acts as the controlled writer to the DuckDB store. This boundary allows Conexus to use a lightweight local-first analytical engine while protecting shared writes through an explicit contract.

8.4 Local-first infrastructure

The platform uses k3s for orchestration, Zot for local Open Container Initiative images, customer-controlled storage, local observability, and local-model execution for bounded automated work. Optional cloud capacity is designed as an operator-armed exception rather than an implicit dependency.

8.5 Ralph repair loop

Ralph is a local-model repair loop for eligible CI failures. It retrieves relevant prior fixes, performs bounded repairs, reruns verification, records results, and escalates failures outside its authority. It is prohibited from weakening tests or controls merely to produce a green pipeline.

8.6 Current authority transition

The present workflow remains human-mediated for Git publication. Agents can produce substantial changes, but unrestricted push authority has not been delegated. The intended future state is evidence-derived authority: routine commit and push operations become eligible only after the system can prove that scope, ownership, verification, evidence, and policy requirements were satisfied.

This distinction prevents planned authority from being represented as already deployed autonomy.

---

9. Security and assurance implications

The Secure Software Development Framework, NIST SP 800-218, recommends integrating secure-development practices into the software-development life cycle. NIST SP 800-218A extends this approach with practices specific to generative AI and dual-use foundation models.

Evidence-Gated Agentic Development operationalizes a compatible principle: increasing AI autonomy should increase the rigor of surrounding controls rather than reduce it.

Relevant evidence may include:

- Test and coverage results
- Mutation and integration evidence
- Static and dynamic application-security results
- Secret scans
- Dependency and license analysis
- Software bills of materials
- Image and infrastructure scans
- Provenance and signatures
- Approval records
- Policy decisions
- Artifact digests
- Exception records

These artifacts can support control assessment and audit preparation. They do not automatically establish organizational compliance, legal sufficiency, regulatory authorization, or safety certification.

A defensible statement is that the platform produces versioned, signed, and inspectable software-delivery evidence that can be mapped into selected assurance profiles.

10. Failure model

A credible agentic platform must distinguish failure categories.

Failure category	Example	Required response
Agent crash	Session disappears during work	Expire lease, preserve checkpoint, permit reassignment
Stale authority	Old session resumes after reassignment	Reject through fencing token
Conflict	Two agents modify the same protected scope	Surface collision and block or coordinate
Tool missing	Required scanner not installed	Fail as TOOL-MISSING
Invalid evidence	Artifact exists but is malformed or empty	Fail as EVIDENCE-INVALID
Infrastructure error	Runner, network, registry, or storage failure	Separate from product failure; retry or escalate
Requirement failure	Tests or policy thresholds fail	Repair within authority or escalate
Policy violation	Agent attempts prohibited operation	Deny and audit
Unknown failure	Classifier cannot establish safe action	Fail closed and request human review

Treating these outcomes as separate states prevents a green-or-red interface from hiding operationally important distinctions.

---

11. Tradeoffs and limitations

11.1 Complexity

The model combines retrieval, coordination, CI/CD, security, infrastructure, observability, evidence, and governance. The resulting platform requires disciplined contracts, migrations, documentation, and operational ownership.

11.2 Verification cost

Broad testing and security profiles consume time, hardware, and storage. Test-impact analysis, caching, parallelism, and tiered profiles may reduce cost, but optimization must not create false-green paths.

11.3 Policy risk

Automated authority depends on correct policy. A defective required-job manifest, exclusion ledger, approval rule, or evidence validator can be as consequential as an application defect. Policy must therefore be versioned, tested, reviewed, and auditable.

11.4 Local infrastructure responsibility

Local-first operation increases control but transfers responsibility for availability, patching, backups, capacity, power, recovery, and hardware failure to the operator.

11.5 Limits of evidence

A signed green bundle proves only that defined controls passed under defined conditions. It does not prove requirements were correct, the threat model was complete, or the software is permanently safe.

---

12. Strategic and career implications

The relative value of software skills changes when generation becomes abundant.

The following disciplines become more important:

- Distributed systems and concurrency
- Information retrieval and knowledge modeling
- Networking and failure analysis
- Secure software development
- Test architecture and mutation testing
- CI/CD and supply-chain security
- Observability and provenance
- Cloud and local infrastructure tradeoffs
- Policy engineering
- Human-machine authority design

The future full-stack engineer is not limited to frontend, backend, and database implementation. The stack increasingly includes models, retrieval, agents, coordination, verification, infrastructure, evidence, approval, and operational learning.

---

13. Public positioning

The clearest public description is:

> Conexus is an agentic software-delivery control plane that gives AI workers shared memory and coordination, independently verifies their output, and requires evidence before consequential work advances.

A concise delivery-model statement is:

> Generate quickly. Verify independently. Promote only with evidence.

Public materials should explain the control philosophy without disclosing sensitive thresholds, permission rules, failure signatures, repair prompts, escalation heuristics, customer evidence, or mechanisms that would materially assist bypass attempts.

---

14. Conclusion

The history of distributed systems is a history of making independent, fallible actors cooperate under imperfect information.

Packet switching established protocol-driven communication. Logical clocks formalized causal order. Byzantine analysis separated agreement from trust. Distributed snapshots made concurrent state recoverable. Remote-call systems exposed the danger of hidden failure semantics. Consensus and lock services formalized authority. Web-scale systems treated failure as normal. Cluster managers reconciled desired and observed state. Distributed tracing reconstructed work across components. Continuous integration reduced the cost of delayed convergence.

AI agent swarms inherit all of these problems.

The correct response is not to demand perfect models or to place a human click after every generated line. It is to build an architecture in which machine-produced work is bounded, attributable, independently evaluated, and unable to cross consequential boundaries without evidence.

The next stage after trunk-based development is not the disappearance of humans. It is humans governing a machine-operated integration system.

Evidence-Gated Agentic Development provides the control model for that transition.

References and evidence basis

1. Steve Crocker, “Host Software,” RFC 1, April 1969.
2. Barry M. Leiner et al., “A Brief History of the Internet,” Internet Society, 1997.
3. Leslie Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” Communications of the ACM, 1978.
4. Leslie Lamport, Robert Shostak, and Marshall Pease, “The Byzantine Generals Problem,” ACM Transactions on Programming Languages and Systems, 1982.
5. K. Mani Chandy and Leslie Lamport, “Distributed Snapshots: Determining Global States of a Distributed System,” ACM Transactions on Computer Systems, 1985.
6. Sun Microsystems, “RPC: Remote Procedure Call Protocol Specification Version 2,” RFC 1057, 1988.
7. Sun Microsystems, “NFS: Network File System Protocol Specification,” RFC 1094, 1989.

8. Leslie Lamport, “Paxos Made Simple,” ACM SIGACT News, 2001.
9. Seth Gilbert and Nancy Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,” SIGACT News, 2002.
10. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “The Google File System,” SOSP, 2003.
11. Jeffrey Dean and Sanjay Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” OSDI, 2004.
12. Fay Chang et al., “Bigtable: A Distributed Storage System for Structured Data,” OSDI, 2006.
13. Mike Burrows, “The Chubby Lock Service for Loosely-Coupled Distributed Systems,” OSDI, 2006.
14. Giuseppe DeCandia et al., “Dynamo: Amazon’s Highly Available Key-Value Store,” SOSP, 2007.
15. Benjamin H. Sigelman et al., “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure,” Google, 2010.
16. James C. Corbett et al., “Spanner: Google’s Globally-Distributed Database,” OSDI, 2012.
17. Diego Ongaro and John Ousterhout, “In Search of an Understandable Consensus Algorithm,” USENIX ATC, 2014.
18. Abhishek Verma et al., “Large-Scale Cluster Management at Google with Borg,” EuroSys, 2015.
19. DORA, “Trunk-Based Development” and “Working in Small Batches.”
20. NIST SP 800-218, Secure Software Development Framework Version 1.1, 2022.
21. NIST SP 800-218A, Secure Software Development Practices for Generative AI and Dual-Use Foundation Models, 2024.
22. Conexus internal documentation snapshot, July 2026, including the system reference, verification remediation contract, federation-and-Brain documentation, feature catalog, infrastructure catalog, and content-production workflow.