

Five laws of agentic delivery

Agentic engineering fails when a persuasive session is mistaken for a delivery system. Five operating laws turn model capability into controlled production work by moving plans, units, depth, model choice and parallelism into inspectable artefacts and gates.

AUTHORED BY

Daniel Johnson & Gary McLean Hall

AI DRAFTING

Astra, Fable & the models of the moment

EVIDENCE CHECKED

15 September 2026. The date tells you when the cited material was reviewed, not when the future agreed to behave.

INVALIDATED BY

When delivery evidence shows that one of the laws creates more risk or toil than it removes.

EXPIRY DATE

The reader's call. Before autonomous theatre is accepted as a production control.

The founders own the argument; AI helped draft the prose. The sources and limits are here so you can challenge both.

Contents

01. [Why laws rather than tips](#)
 02. [Law one: the plan lives on disk; the context is a viewport](#)
 03. [Law two: one unit equals one session equals one commit](#)
 04. [Law three: depth is a gate property, not a prompt property](#)
 05. [Law four: spend tokens by tier—frontier proposes, humans authorise, workhorse executes, gates verify](#)
 06. [Law five: parallelise across artefact boundaries; keep orchestrator context flat](#)
 07. [How the laws work together](#)
 08. [A minimal session card](#)
 09. [Counterarguments and limits](#)
 10. [What leaders should ask to see](#)
 11. [Conclusion](#)
 12. [References](#)
-

These laws are an operating proposal, not experimentally established laws of software delivery. The review date records an editorial and source check. Research is cited for the narrower claims it supports; the session cards and delivery examples are constructed exhibits, not measured customer results or a substitute for premise-author review.

Why laws rather than tips

An AI coding session can be impressive while the resulting product is incoherent. The model can produce thousands of lines, answer every objection and leave the visible screen green. None of those facts establishes that the agreed job is complete, that the architecture still holds, or that another person can safely operate what was built.

A delivery claim needs a boundary

Activity

Code, narration and apparent autonomy are visible outputs.

Acceptance

A bounded artefact survives a named challenge and preserves its evidence.

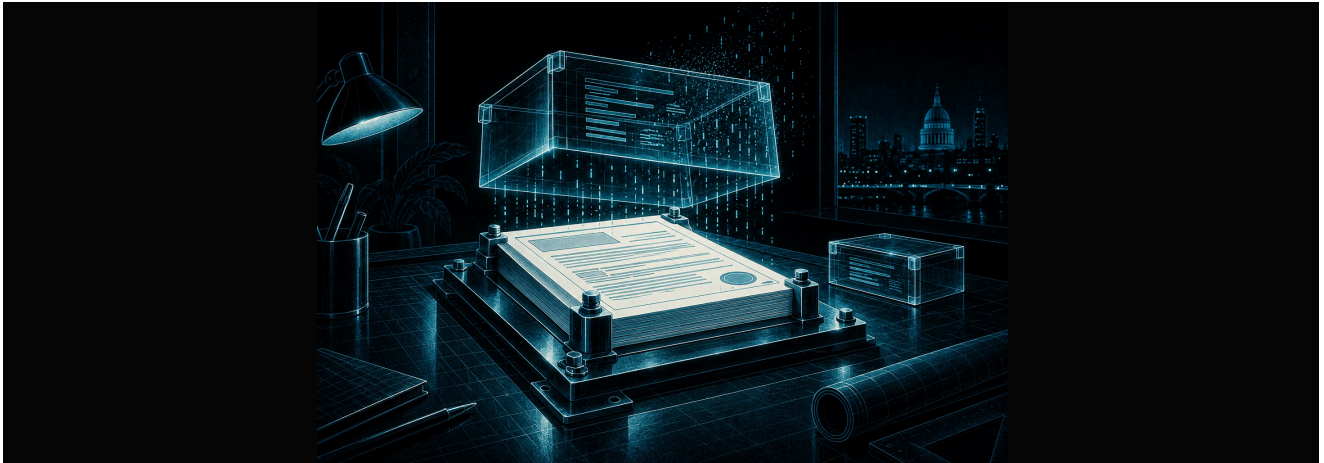
Agentic delivery therefore needs an operating system, not a better incantation. The five laws below emerged from building software with multiple model roles, persistent specifications and adversarial gates. They are propositions to test, not natural laws. Their purpose is to make failure observable.

The control problem appears at handoffs. A session can understand its own choices while leaving the next worker unable to reconstruct them. A reviewer can approve a patch without knowing that its tests were rewritten to match the implementation. An orchestrator can report progress while two workers are relying on incompatible versions of the same contract. None of these failures requires an unintelligent model. They arise because authority, state and acceptance are implicit.

The laws make those handoffs inspectable. They should be judged by whether they reduce ambiguity and rework, not by how many process artefacts they produce. If a rule adds ceremony without improving a decision, its implementation needs attention. If bypassing it removes the only record of what was agreed or tested, the apparent saving is being borrowed from future investigation.

Law one: the plan lives on disk; the context is a viewport

A conversation is temporary working context. The plan, scope and evidence must be versioned artefacts outside it.



Long-context research shows why. Models do not necessarily use every part of a large prompt equally; relevant information can be missed when buried among other material. Sessions also end, summaries compress nuance and concurrent agents see different slices of state. If the only authoritative plan lives in a chat, the project has no stable source of truth.

The research boundary matters. *Lost in the Middle* tested information use in document question answering and key-value retrieval, finding sensitivity to where relevant information appeared. It did not compare disk-based plans with chat-based software programmes, and it does not establish the behaviour of every later model. Durable plans are our engineering response to an authority and continuity problem; the study supplies a reason to challenge assumptions about context use. [Liu et al., 2023](#).

“On disk” is shorthand for durable, reviewable state. A repository should contain the product brief, obligations, architecture decisions, schema, unit plan, acceptance criteria, test commands and current evidence. A session card points a model at the specific unit and the minimum surrounding context required to execute it.

This is not bureaucracy for its own sake. It reduces the amount each model must infer. It also lets a human inspect what the system believes before authorising more work.

A practical test is simple: could a fresh engineer, without the chat transcript, identify the current objective, the boundary of the next change and the evidence required to accept it? If not, the plan does not yet live on disk.

Durability alone is insufficient. An outdated plan is durably wrong. The unit record should distinguish intended state, observed state and unresolved decisions, with the evidence revision attached to the observed state. A worker must not read an aspiration such as “migration complete” as proof that migration was exercised. The record should point to the rehearsal or state clearly that the claim remains unverified.

This is particularly useful after interruption. Resumption begins by checking the current artefact and repository state against the last accepted record. It does not begin by trusting the last confident message. Uncommitted work can be valuable and should be preserved, but its presence does not establish that the unit passed. A fresh session needs to know what has changed, what was actually checked and which checks are still owed. The plan becomes an index of authority and evidence rather than a diary of enthusiasm.

Law two: one unit equals one session equals one commit

A unit is a coherent, independently verifiable change. It should be small enough for one agent session to understand and complete, and large enough to leave the system in a meaningful state. The accepted result becomes one commit.

What belongs in the commit

Coherent unit

The accepted implementation with its supporting checks or evidence reference.

Review boundary

Its scope still matches the unit the session was authorised to deliver.

This law creates three boundaries at once. The session limits context. The unit limits scope. The commit preserves provenance and rollback. When a unit sprawls across many unrelated concerns, failures become ambiguous. When a session accumulates many units, stale assumptions compound. When several units are combined in one commit, review and reversion become archaeological work.

The law does not require every typo to have its own ceremony. Nor does it mean a commit is correct because it is small. It means that the smallest useful claim—“this capability now exists and these gates support it”—has a matching technical artefact.

Modern code-review research at Google describes review as a tool-mediated process operating over changes, not an abstract judgement over a whole codebase. The same principle matters more with agents: inspectability is a property of the change boundary.

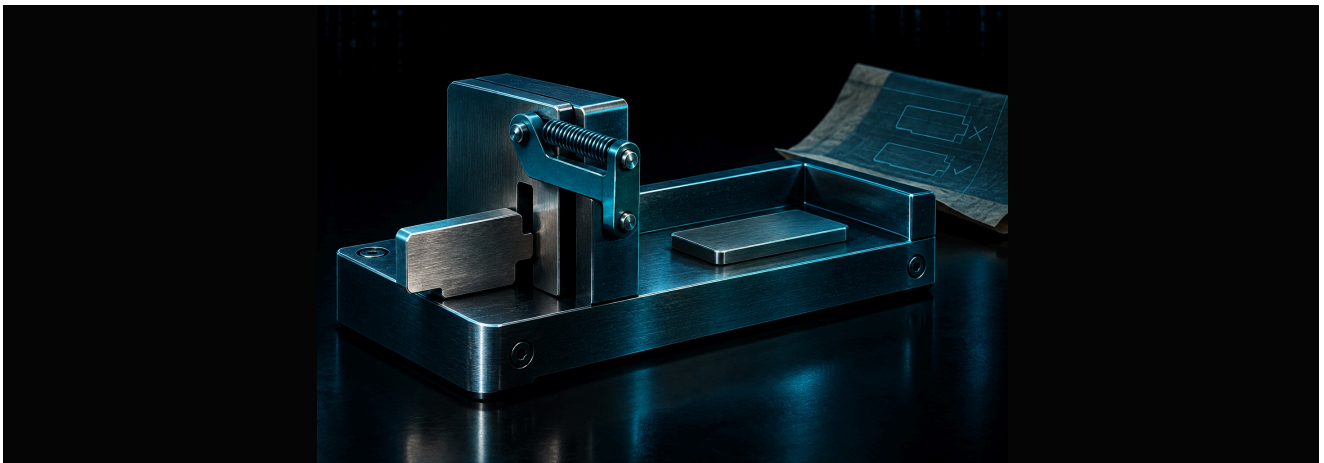
Unit design should therefore state inputs, files likely to change, non-goals, acceptance criteria, commands and evidence. If implementation reveals that the unit contains two independent obligations, split it before accumulating more work.

Consider an illustrative change to notification retries. A useful unit might include the retry state, persistence, worker behaviour, user-visible failure status and tests proving that retries do not duplicate the business event. Splitting it into “add a column”, “write a worker” and “add a status badge” could leave several individually tidy commits without an independently usable capability. Conversely, including a new authentication design because the same worker noticed it would destroy the unit boundary.

The right size follows the acceptance claim and dependency structure, not a fixed line count. Some changes need preparatory units that establish contracts without exposing the final capability. Those units should state their narrower claim honestly. A coordinated release can combine several accepted units while preserving their separate provenance. The law is a discipline of coherent change, not a demand that every meaningful system improvement fit into one small diff.

Law three: depth is a gate property, not a prompt property

“Be thorough” is not verification. Neither are “production-ready”, “enterprise-grade” or “think harder”. A model can respond to those phrases with more code and more confident prose while leaving the same missing depth.



Depth must be represented by gates that can fail. If concurrency matters, run concurrent operations and assert invariants. If migration matters, exercise old data and rollback. If permissions matter, attempt forbidden actions. If accessibility matters, test keyboard and assistive-technology paths. If operations matter, restart the service, expire a credential and observe recovery.

NIST's Secure Software Development Framework treats secure development as a set of practices integrated through the lifecycle, including tracking security requirements, risks and design decisions. The useful idea is broader than security: quality attributes must become work products and verification activities, not adjectives.

A depth signature is the minimum set of observable dimensions required for the particular unit. A static content change may need rendering, links, accessibility and responsive checks. A payment-state change may need idempotency, concurrency, audit, recovery and reconciliation. Applying the same checklist to both is cargo culting; applying no explicit depth model leaves the decision to the model's sense of plausibility.

The test of a gate is adversarial: can it fail when the promised property is deliberately broken? A test suite that cannot detect a seeded relevant fault is decoration.

Gates should cover the durable effect as well as the visible response. In the notification example, a test that sees "sent" on screen can miss two underlying deliveries or a lost retry record. The challenge should interrupt the worker at a relevant boundary, replay the operation and inspect the resulting state and emitted effects. The evidence then supports a particular failure claim instead of the vague proposition that the feature worked once.

There is also a governance boundary around the gate itself. Agents may discover that an assertion is wrong or a fixture no longer represents the contract. They should be able to propose a correction, but the change must identify which obligation remains protected and demonstrate that a relevant fault still fails. Otherwise acceptance becomes a negotiation between the producer and its own test suite. The process needs a record of why the oracle changed, not a blanket prohibition on ever improving tests.

Law four: spend tokens by tier—frontier proposes, humans authorise, workhorse executes, gates verify

Model capability is not one undifferentiated resource. Architecture, ambiguity resolution and cross-system judgement benefit most from strong reasoning. Mechanical implementation, search, transformation and bounded test writing can often be handled by cheaper or local models. Acceptance should not be delegated to either class; executable gates decide whether the artefact satisfies the claim.

Role allocation

Ambiguity

Frontier reasoning proposes; a human owns consequential meaning.

Mechanical work

A bounded worker executes; independent gates decide technical acceptance.

This is an economic and a control principle. Using the most expensive model for every keystroke wastes capacity. Using a weaker model to settle an ambiguous system boundary creates expensive rework. Asking the same model that wrote a change to certify it confuses production with evidence.

The research record warns against universal productivity claims. METR's 2025 randomised trial found experienced developers working in familiar open-source repositories were slower with the tested AI tools. Peng and colleagues found faster completion on one bounded programming exercise using GitHub Copilot. The reasonable conclusion is task allocation: measure where a model helps, keep human judgement on consequential decisions, and make verification independent of self-report.

These results concern different tasks and interventions. METR studied 16 experienced contributors undertaking 246 real issues in their own repositories, with access to early-2025 tools assigned by issue. Peng and colleagues studied an HTTP-server implementation task. Neither directly tests a frontier/workhorse allocation or these five laws. Their disagreement warns against transferring a result across workflows without a local comparison. [METR, 2025](#); [Peng et al., 2023](#).

A tiered system might use a frontier model to compile a unit and resolve contradictions, a workhorse model to implement it, deterministic tooling to format and analyse it, and a separate adversarial review to find narrative or architectural drift. The exact vendors can change. The roles remain.

Tiering requires escalation criteria. A workhorse assigned a stable transformation should stop treating it as mechanical when sources disagree, the public interface must change or the unit reveals an unmodelled domain rule. The escalation should return the conflicting facts and a bounded decision, not the entire transcript. A frontier session can propose a resolution for the accountable owner to approve, then return the revised contract while the mechanical worker retains a narrow scope.

Cost accounting must include that loop. A cheaper model with a low acceptance rate can consume more review and repair than a stronger model used directly. Equally, a frontier model may waste capacity on deterministic transformations that ordinary tooling can perform. Compare cost per accepted unit, including setup, failed attempts and verification. Track the kind of work and the starting specification quality so a reduction in cost is not purchased by accepting shallower evidence or transferring uncertainty to the reviewer.

Law five: parallelise across artefact boundaries; keep orchestrator context flat

Parallel agents do not create free throughput. They create coordination cost. Two agents editing the same file or depending on an unsettled interface can produce conflicts faster than one agent produces code.

Parallel work needs a settled seam

Shared contract

Resolve ownership and interface meaning before consumers diverge.

Worker return

Artefact paths, evidence and unresolved findings, not an expanding transcript.

Safe parallelism follows artefact boundaries: separate packages, independent content files, isolated migrations, bounded research questions or tests that do not redefine the implementation contract. The orchestrator should retain the plan, dependencies and acceptance state without absorbing every intermediate transcript.

Flat context means summaries and artefacts flow upward, not entire conversations. Each worker returns the changed paths, commands run, evidence, unresolved risks and any discovered change to the contract. The orchestrator reconciles those facts against the on-disk plan.

This resembles ordinary distributed systems. Work needs ownership, interfaces, idempotent operations and an authoritative state. Adding intelligent workers does not repeal coordination theory.

The practical constraint is dependency ordering. If one unit defines a schema used by three others, complete and gate the schema first. Then parallelise the consumers. If two agents discover conflicting requirements, stop the affected branch and resolve the contract rather than letting both interpretations harden into code.

Artefact ownership is necessary but not sufficient. Workers can edit separate files while competing over one semantic decision. A frontend and backend may both implement a field called `status` and still disagree about whether “submitted” includes failed notification delivery. Parallel execution is safe only after the shared meaning is fixed or an explicit reconciliation step is part of the plan.

The orchestrator should therefore maintain a compact dependency ledger: the contract revision consumed, the owner of each unit, its current acceptance state and evidence returned. If a shared contract changes, affected consumers are identified and rechecked.

Completion messages are observations, not automatic transitions to accepted. The orchestrator verifies that the reported artefact exists and the evidence belongs to that revision. This keeps central context small without making central authority superficial.

How the laws work together

The laws form a control loop.

One connected acceptance record

Start

Durable authority specifies the bounded unit and its depth.

Finish

A matching change, refused faults and an explicit acceptance disposition.

The durable plan identifies the system and its next units. Unit boundaries constrain each session and commit. The depth signature defines what completion means beyond the happy path. Model tiers allocate judgement and mechanical effort. Artefact boundaries permit parallel work without fragmenting authority.

Remove one law and a familiar failure appears:

- no durable plan: the system becomes whatever the latest session remembers;
- no unit boundary: change sets become unreviewable;
- no depth gates: “done” means persuasive output;
- no tiering: cost rises or judgement is misplaced;
- no parallelism discipline: conflicts and architectural drift consume the apparent speed-up.

The combined system does not require a particular agent framework. It can operate with one developer and one assistant or with an orchestrated fleet. The variable is volume. The control problem is the same.

An illustrative notification-retry change shows the interaction. The plan names the notification obligation and current domain contract. One worker implements the bounded retry unit; another can independently review a documentation artefact once the interface is settled. The depth signature demands interruption, replay and durable-state checks. A cheaper model performs mechanical fixture work, while an ambiguous policy question returns to a stronger reasoner and the domain owner. The accepted implementation and its evidence become a coherent commit.

If replay reveals duplicate delivery, the unit remains incomplete even if the interface looks finished and every worker has returned. If the domain owner changes what counts as a duplicate, the specification and affected gates change before work resumes. These are

different outcomes. The laws make the distinction visible, which is their practical value: they prevent volume, activity and conversational confidence from standing in for a supported completion claim.

A minimal session card

Each unit should begin with a small, explicit card:

Interrupted is a state

The unit is complete only when resuming delivery preserves the accepted referral and exposes unresolved provider state.

Challenge A	Interrupt before the provider accepts; resume persisted work.
Challenge B	Interrupt after acceptance, before local acknowledgement.
Must survive	The same referral version and one submission event.

Constructed session card · unsupported provider guarantees require a domain decision.

1. Objective: the observable capability or decision.
2. Boundary: what may change and what must not.
3. Sources: authoritative files and decisions.
4. Depth signature: failure surfaces that must be exercised.
5. Gates: exact commands and acceptance thresholds.
6. Handover: artefacts, evidence and known risks to return.

The card should not restate the whole programme. It should point to durable sources. The agent's first job is to test whether those sources agree. If they do not, implementation has not started; ambiguity resolution has.

A session card should include a concrete stop condition. "Finish the feature" gives the worker no reliable way to distinguish discovery from completion. "The specified retry scenarios leave one business event, preserve visible failure state and pass the named checks" describes an observable boundary. The card should also identify the evidence expected at handover: changed paths, command results, relevant versions and unresolved limitations.

The worker's opening audit should be short but real. Confirm that named sources exist, that their contracts agree and that the starting code is the state the unit expects. This avoids spending the session implementing against an imaginary baseline. At the end,

compare the delivered artefact with the same card. Newly discovered work should be recorded explicitly; it should not disappear because it complicates a positive status report, nor be silently absorbed until the unit loses all shape.

Session-card exhibit: recover an interrupted notification

The following is a constructed example. Its identifiers describe a proposed unit, not files or commands claimed to exist in a customer repository. A real card must replace these references with reachable, versioned artefacts before execution begins.

Card field	Content the worker receives
Objective	Resume a pending notification after worker failure without creating a second referral submission
Authority	Accepted submission contract, delivery-state definition and provider retry contract, each at its named revision
Owned change	Delivery worker, persistence adapter and the tests exercising their boundary
Preserved invariant	A retry cannot change the accepted referral version or create another submission event
Required challenge	Interrupt before provider acceptance and after acceptance but before local acknowledgement; resume from persisted state
Observation	Submission identity, delivery-attempt history, pending or confirmed status and emitted provider request
Acceptance	Named repository checks pass; the deliberately broken retry path is rejected; evidence refers to the delivered revision
Escalation	Provider semantics cannot support the required guarantee, or the domain meaning of duplicate delivery is undecided

The worker first checks that the existing implementation and fixtures represent those states. Suppose it discovers that the provider offers no way to distinguish an accepted request from one lost before acceptance. That is a contract problem, not a request for more ingenious retry code. The worker returns the observed provider behaviour and the exact guarantee it cannot support. The owner can decide whether to accept occasional duplicate notification, introduce reconciliation or change the service boundary. The referral's own single-submission invariant can remain intact throughout that decision.

If the contracts are coherent, implementation proceeds in the owned artefacts. The verifier then runs the challenge against the resulting revision and against a controlled faulty version. The faulty version might create a new submission when delivery resumes. A rejection demonstrates sensitivity to that mechanism. It does not prove that every possible provider failure is covered, and the report should retain that limitation.

At handover, the worker returns the changed paths, resulting revision or patch, exact commands, observed outcomes and any incomplete checks. “All tests pass” is insufficient when the relevant check never ran because its environment was unavailable. A truthful incomplete result preserves more value than an accepted label that the next worker must disprove. The orchestrator verifies the artefact against the card before marking it accepted.

The commit should contain the coherent implementation and its supporting tests or durable evidence reference. It should not bundle an unrelated interface redesign encountered during the session. If the accepted change is later reverted, the organisation can identify which guarantee it is withdrawing and which dependent work needs attention. That is the practical connection between session scope, change history and operational control.

Counterarguments and limits

Strict unit boundaries can create overhead. Some exploratory work cannot be specified cleanly in advance. Creative design benefits from looser iteration. Large refactors sometimes cross many artefacts by necessity.

Discovery can finish without a patch

Valid output

A cause, a map or a defensible next decision.

Wrong incentive

Requiring a code commit when investigation did not call for a change.

The answer is not to pretend all work is deterministic. Discovery can be a unit whose output is a decision or a map rather than production code. An architectural spike can be explicitly disposable. A large refactor can be divided by observable seams even when the final transition is coordinated.

The laws also do not solve bad requirements. A precise plan can encode the wrong goal. Human product judgement, domain expertise and user research remain upstream obligations. Mechanical evidence can show that a system implements a rule; it cannot decide whether society wanted that rule.

An emergency is another important limit. A failing live service may require containment before a full implementation contract can be written. The doctrine should support a narrow emergency authority: state the observed harm, permitted intervention and immediate verification, preserve the evidence, then reconcile the durable record after containment. Inventing a complete plan retrospectively would corrupt the incident history. Refusing a necessary authorised containment action solely because a session-card field is incomplete would mistake the control's format for its purpose.

The laws also cannot guarantee independent review merely by assigning another agent. Reviewers can inherit the same mistaken source, accept the same generated expected values or optimise for the same superficial gate. Independence requires a different route to challenge: a domain counterexample, observed state, a meaningful mutation or a reviewer who questions the acceptance boundary. Model diversity may help, but vendor names alone do not establish evidential independence.

The strongest objection is that disciplined artefacts can create false precision. A team may faithfully implement an agreed but harmful workflow, or optimise a measurable gate while neglecting what users actually need. The answer is to keep product legitimacy outside mechanical acceptance. Domain review and user evidence can change the target; the delivery system then makes that change explicit and traces its consequences.

There is also a risk of process rigidity. During investigation, the right unit may be “find the cause and identify a defensible next decision”, with no production patch expected. Requiring a code commit would reward unnecessary change. Likewise, preserving a coherent unit does not require keeping a failing attempt alive forever. A worker may return evidence that the plan is defective. The important rule is honest classification of the result and preservation of useful findings, rather than forcing every session into the appearance of success.

What leaders should ask to see

An organisation adopting agentic engineering should be able to show:

Inspect resumability

New worker receives

The plan, accepted contract, changed paths and recorded checks.

Evidence of a gap

The previous conversation is required to reconstruct what “done” meant.

- a versioned plan outside vendor chat history;
- a map from business obligations to units and evidence;
- commits whose scope matches those units;

- gates that have been proven capable of failing;
- a record of who authorised consequential decisions and which model outputs informed them;
- cost and cycle-time evidence by task type; and
- a recovery path when generated work is rejected.

These are ordinary engineering controls adapted to a faster producer. They are intentionally less theatrical than autonomous-agent demos. The point is not to watch software type itself. The point is to make a dependable product arrive with less human toil and more inspectable evidence.

Adoption can itself be treated as an evidence exercise. Compare units with similar obligations and acceptance depth, recording preparation, execution, review, rework and handover effort. Inspect whether failures are discovered earlier and whether another worker can resume without reconstructing the conversation. Do not reward compliance by counting session cards or commits; those are means of making work inspectable. If the process creates many small artefacts but still leaves responsibility and acceptance unclear, it has preserved the shape of the laws while missing their purpose. A useful operating doctrine should survive criticism from its own delivery data.

Conclusion

Agentic delivery is not a conversation with a clever model. It is a production system in which models occupy bounded roles.

The plan lives durably. Units constrain change. Depth is executable. Capability is spent where judgement warrants it. Parallelism follows stable seams. Gates—not confidence, token count or visual activity—decide whether work survives.

Those laws do not make inference deterministic. They make its contribution governable.

References

- Liu et al., [Lost in the Middle](#), 2023.
- METR, [Measuring the impact of early-2025 AI on experienced open-source developer productivity](#), 2025.
- NIST, [Secure Software Development Framework 1.1](#), 2022.
- Sadowski et al., [Modern Code Review: A Case Study at Google](#), 2018.
- Google, [Site Reliability Engineering: principles](#), 2016.
- Peng et al., [The Impact of AI on Developer Productivity: Evidence from GitHub Copilot](#), 2023.