

CONTRACT-ENFORCED, EVIDENCE-DRIVEN SOFTWARE DELIVERY

The CEED Framework™

AI-Native Software Delivery Governance

Official positioning

Contract-enforced, evidence-driven software delivery. An AI-native operating model for authorizing what a code change may touch and recording what happened after it shipped.

Author	Viktor Barakoski
Document status	Whitepaper — Public Review Draft
Date	September 25, 2026
Ownership notice	The CEED Framework™. Copyright © 2026 Viktor Barakoski.
Official resources	ceedframework.com

A Public Review Draft of the CEED Framework whitepaper

Your pipeline checks whether the change is safe to run.

The CEED Router checks whether the change stayed within its approved authorization before it can merge or ship.

Authorization decided before code generation. Evidence before release. A record after production.

Publication information

Licensing

This Public Review Draft is released under CC BY 4.0. The documentation and data artifacts intended for the Version 1.0 package, including the CEED Specification, schemas, CEED Path Pattern Profile 1.0, tool catalog, and conformance materials, are intended to be released under CC BY 4.0. Source code for the CEED Router is intended to be released under Apache License 2.0. These licenses do not grant endorsement, official status, trademark rights, or a right to claim conformance by the CEED Project.

Trademark notice

The marks CEED™ and CEED Framework™ are trademarks of Viktor Barakoski. Use of the CEED Specification, the CEED Router, or other components under the licenses described above does not grant any right to use these marks.

Author independence and personal views

CEED Framework™ is an independent framework authored by Viktor Barakoski and developed using personal time and personal resources. The views in this document are the author's personal views, and no confidential, proprietary, restricted, or non-public materials were used in its development.

Recommended citation

Barakoski, Viktor. (2026). The CEED Framework: AI-Native Software Delivery Governance. Whitepaper, Public Review Draft, September 25, 2026. Official resources: ceedframework.com.

Status of this document

This document is the Public Review Draft of the CEED Framework whitepaper. It explains the framework as a domain-specific standardization proposal for AI-native software delivery and presents the problem, operating model, enforcement pattern, adoption shape, and expected control value. It is not the Version 1.0 release and may change before that release. It is not normative. For the intended Version 1.0 release, the CEED Specification and Change Passport Schema will govern where explanatory documents differ. References to Version 1.0 in this draft describe the current target release unless stated otherwise. This draft does not claim that the CEED Framework is already an adopted industry standard.

Evidence status in this Public Review Draft

Category	Current status
Defined in the current working package	Delivery Contract and Change Passport semantics; schemas; CEED Path Pattern Profile 1.0; decision algorithms; worked lifecycle; Passport-native reporting definitions; Tool Verification Matrix; Conformance Pack.
Exercised in the current conformance material	Schema validation currently exercises 14 named positive and 15 named negative examples. Separately, the conformance runner executes 80 behavioral vectors and five golden reports - 85 executable checks in total - against the current packaged rules.
Not yet released as Version 1.0	The Version 1.0 Specification, Conformance Pack, CEED Router, schemas, worked example, and supporting evidence documents are not yet public. They are intended to be released together after the remaining review decisions and normative corrections are incorporated and the release checks are complete.
Not established by current evidence	Independent implementation interoperability; production availability, security, latency, and scale; organization-specific adoption effort; and measured cost, safety, audit, or delivery benefits.

Executive summary

AI coding agents can produce working changes faster than engineering teams can review them. They can also make changes that were never part of the approved work. A candidate may compile, pass every test, and still modify a billing configuration, an authentication rule, a shared schema, or an infrastructure file that nobody authorized it to touch.

Delivery pipelines already perform many checks before code runs: tests exercise behavior, scanners inspect dependencies, branch rules control merge, deployment systems control release, and observability watches production. What those controls do not necessarily carry is one machine-readable statement of what this particular change was authorized to modify before the code existed.

The CEED Framework addresses that gap with two records and one enforcement component. A CEED Delivery Contract records the approved requirement, allowed scope, protected scope, required merge evidence, risk tier, release rule, and schema version. A CEED Change Passport preserves the decisions and production knowledge that follow. The CEED Router compares the exact change and release evidence with the approved Contract at batch admission where applicable, at the merge gate, and at each required deployment checkpoint. It writes every decision to the Change Passport.

The model deliberately moves authority outside the coding agent. A prompt, repository instruction file, or IDE rule can guide an agent, but it is not the same as an approved authorization enforced by the delivery pipeline. The agent may draft a Contract and use it while working; it cannot issue, approve, or widen its own authority. People decide intent, boundaries, and exceptions. Machines perform the repeatable production work and deterministic checks.

The CEED Framework does not replace the delivery stack. The requirement remains in the organization's existing work-management system. Git remains authoritative for commits and changed paths. CI and security tools continue to produce test and scan evidence. Deployment platforms execute the rollout. Observability and incident systems remain authoritative for production facts. The framework gives those systems shared per-change semantics and keeps their decisions tied to one change history.

This matters for both control and usability. Teams already pay for scope discipline through ticket writing, prompt instructions, manual diff review, failed merge groups, release checks, and later reconstruction. The framework moves the reusable part of that work into one approved record and lets the Router perform the comparisons thereafter. The framework still requires setup and human approval, and Version 1.0 does not claim a measured net saving. Its practical proposition is narrower: write the authorization once, enforce it consistently, and stop asking every later reader to reconstruct it from scattered systems.

The same record creates a stronger structural basis for audit and learning. A Change Passport shows which Contract version governed a candidate, which evidence was checked, how the release advanced, and what was later learned in production. The CEED Router calculates defined local measures from those Passports without making network calls. Optional cross-organization comparison, when available, uses aggregate contributions rather than the underlying Contract and Passport records.

The central claim

The CEED Framework does not invent new policy, provenance, rollout, or enforcement primitives. It proposes one standard way to bind an approved requirement, repository-path authorization, evidence, release decisions, and evolving production knowledge into one per-change lifecycle.

How to read this document set

The planned Version 1.0 document set separates explanation, example, normative rules, evidence, and executable tests. The companion documents below are pre-release and not yet public.

Document	Purpose
CEED Version 1.0 - Start Here	A concise entry point to the problem, model, evidence status, package map, and conformance material.
This Whitepaper — Public Review Draft	Explains why the model exists, how the lifecycle works, what adoption changes, and where value is expected.
CEED in Practice	Follows one change through the lifecycle and shows the Delivery Contract, Change Passport entries, gate decisions, and local reporting output in stored form.
CEED Specification and Change Passport Schema	Defines the fields, path profile, algorithms, reporting rules, and conformance requirements normatively.
CEED Tool Verification Matrix	Provides the dated primary-source evidence behind the adjacent-tools analysis and identifies possible implementation roles.
CEED Conformance Pack	Provides schemas, examples, semantic checks, behavioral vectors, report fixtures, and golden outputs for executable agreement testing.

Reading paths by role

Reader	Primary question	Where to start
CEO, CIO, and engineering leadership	What problem does the framework solve, and what changes in the model?	Introduction; Chapters 1, 2, 8, and 9; Closing
Platform engineering	What must be added to one repository and delivery path?	Chapters 3, 5, and 6
Staff engineering and architecture	Are the records, identifiers, and decisions deterministic?	Chapters 3, 4, 5, and 7
Supply-chain security and DevSecOps	How do authorization, evidence, provenance, and enforcement relate?	Chapters 3, 5, 7, and 9
Release engineering and SRE	How are rollout checkpoints and production outcomes handled?	Chapters 3, 4, 5, and 6
Audit, risk, and governance	What evidence is preserved, what remains external, and what does the record not prove?	Chapters 4, 7, and 9
Tool and standards maintainers	Where can existing systems produce, carry, enforce, or consume CEED semantics?	Chapter 7

Start Here is planned as the entry point to Version 1.0. When the package is released, CEED in Practice will show the complete stored lifecycle, while the Specification and Conformance Pack will govern implementation behavior. Until then, this Public Review Draft is explanatory and not normative.

Contents

Introduction — The authorization decision moves upstream.....	8
1. The Scope-Control Gap.....	9
2. The CEED Operating Model.....	11
3. The CEED Delivery Contract.....	13
4. The CEED Change Passport.....	17
5. The CEED Router.....	20
6. What a Team Sets Up.....	22
7. Existing Tools and the Missing Common Language.....	24
8. What Adoption Adds, Moves, and May Reduce.....	27
9. Assurance, Conformance, and Learning.....	31
Closing.....	35
References and companion documents.....	35

Introduction

The authorization decision moves upstream

Software delivery has become much faster at producing code. It has not become equally fast at deciding what the code was authorized to change. That difference matters when AI coding agents can create, revise, test, and prepare changes at a volume that human review was never designed to absorb.

A change can pass every quality check and still be wrong in a different way. An agent may update a billing file while fixing a ticket API, alter an authentication rule while changing a user preference, or modify a shared schema because doing so makes the requested feature easier to complete. The code may work. The problem is that nobody authorized that part of the change.

Teams already make authorization decisions. They agree on the requirement, identify the expected area of impact, call out systems that must remain untouched, decide what evidence is needed, and choose how the change should reach production. Today those decisions are often split across a tracker, a planning conversation, a prompt, a pull request, and a reviewer's memory. The decision exists, but the delivery system cannot reliably enforce it as one decision.

The CEED Framework moves that decision upstream. Before implementation begins, the team approves a Delivery Contract for the change. The CEED Router then evaluates the exact Git candidate presented at the gate: it checks which repository paths changed, whether any protected or unlisted path was touched, whether the required evidence belongs to that candidate, and whether the approved release rule is being followed. It does not inspect and approve the internal contents of every file, and it does not monitor each workspace edit while the agent is working.

The Change Passport preserves what follows. It records the Contract version that was issued, the candidate evaluated, the evidence checked, the release checkpoints approved or blocked, and the production outcome later recorded. When new facts appear, they are appended rather than used to rewrite the past.

What the cover claim means

The CEED Framework places the authorization decision before code generation and enforces it before merge and while a rollout advances through its required checkpoints. It checks whether the exact candidate stayed inside the approved repository-path boundary and presented the required evidence. It does not decide whether the code inside an allowed file is correct, whether the approved boundary was wise, or whether the surrounding organization has sound engineering discipline. Those judgments remain with the people and controls that already own them.

This division of work defines the model's AI-native role. AI agents can produce much of the routine delivery output. Machines enforce the deterministic parts of the approved file scope and evidence rules. People remain accountable for intent, scope, exceptions, and material risk. A team can adopt that discipline while engineers still write most of the code and retain it as AI takes on a larger share of production work.

The paper now follows one cumulative argument. Chapter 1 defines the scope-control gap. Chapter 2 explains the operating model and responsibility split. Chapters 3 and 4 explain the Delivery Contract and Change Passport. Chapter 5 shows how the Router enforces them. Chapter 6 describes what a team sets up. Chapter 7 explains how the CEED Framework works with existing tools and why common semantics and conformance matter. Chapter 8 examines adoption work and likely friction. Chapter 9 covers assurance, regulatory relevance, conformance, measurement, and the limits of Version 1.0.

CHAPTER 1

The Scope-Control Gap

AI increases the amount of software a team can produce. It also increases the importance of knowing what each change was actually authorized to touch.

1.1 What recent research is testing

Recent research has begun to measure out-of-scope behavior directly rather than treating it as a hypothetical risk. OverEager-Bench is a 2026 research benchmark created by seven authors affiliated with several universities and Quantstamp. It tests coding agents on benign software tasks and records actions that go beyond the work the user requested. Across 500 validated scenarios and roughly 7,500 runs involving four coding-agent products and six base models, the reported out-of-scope rate ranged from 5.4 to 27.7 percent in the more permissive agent frameworks, compared with 0.2 to 4.5 percent in the ask-to-continue framework tested. These are benchmark results, not production incident rates, but they show that the behavior is measurable and not confined to one model or one agent product. [1]

The same study also shows that explicit scope language helps. When the researchers removed consent or scope declarations from otherwise paired scenarios, the out-of-scope rate increased by 11.9 to 17.2 percentage points across the shared base models. In one reported condition it rose from 0.0 to 17.1 percent. The lesson is not that prompts are useless. It is that the boundary itself is doing important control work, and an agent should not be expected to infer that boundary reliably when it has not been stated. [1]

Two other studies reach the authorization problem from different directions. AuthBench asks whether coding agents can infer least-privilege permissions for realistic terminal tasks. It finds that agents can omit access they genuinely need while also granting access they do not need, and that more reasoning time does not reliably correct the mismatch. [2] Capability Gates examines tool-using agent frameworks and shows that exposing a tool to an agent is not the same as authorizing every concrete parameter value the agent may send on every call. [3]

Taken together, the studies support a narrow operational conclusion: the agent that performs the work should not be the only place where the boundary is inferred, interpreted, or enforced. None of the studies evaluates CEED, and they do not prove that a Delivery Contract will reduce incidents. They do show why explicit authorization deserves to exist outside the agent and to be checked independently.

1.2 A prompt can guide the agent; it cannot serve as the whole control

Teams already use prompts, repository instruction files, IDE rules, and context engines to steer AI coding tools. These controls are useful because they tell the agent what to do and what to avoid. They can reduce unnecessary exploration and prevent some mistakes before a candidate reaches the pipeline.

They do not, by themselves, answer the governance question. A prompt may change between attempts. Different people may state the boundary differently. The agent may misunderstand or ignore the instruction. The instruction may not be approved by the person accountable for the system. Even when the prompt works, the merge or deployment platform may have no durable, machine-readable record of the boundary it was supposed to enforce.

The CEED Framework therefore treats agent guidance and delivery authorization as related but different things. The same Delivery Contract can be shown to the agent so it has useful boundaries while it works, but the authoritative check happens outside the agent against the exact Git candidate and release evidence presented at the gate.

Guidance is not authorization

An instruction inside an AI tool can help the agent stay within scope. An approved Delivery Contract tells the delivery system what the change is allowed to do and gives the pipeline something independent to enforce.

1.3 Human review cannot be the only scope-control mechanism

Pull-request review depends on an engineer understanding the proposed change before it merges. That capacity is bounded by headcount, available time, and the size of the diff. AI can increase both the number of changes and the speed at which they arrive without increasing the number of people available to review them.

AI assistance can make review easier. It can summarize diffs, generate tests, highlight risks, and suggest comments. Those capabilities improve individual steps, but they do not create one approved statement of what the change was allowed to touch. A reviewer still has to compare the generated candidate with an approved file scope that may be scattered across a ticket, prompt, architecture discussion, and personal knowledge.

The CEED Model does not remove code review. It removes one question that should not have to be answered from the code alone: was this file part of the approved change? The Router answers that question deterministically from the Contract and the Git diff, allowing reviewers to spend their judgment on design, maintainability, behavior, and risk.

1.4 Scattered authorization becomes expensive when someone must reconstruct it

When an unauthorized change reaches production, the organization needs more than the final diff. It must establish what the change was approved to do, which file scope and protections were in force, who approved them, what evidence supported the decision, how the release proceeded, and what was later observed.

Regulated and audited organizations have long been expected to show approved and traceable change control. AI does not create that obligation. It changes the producer and the volume. A machine can now create more changes than a review process built around human production can comfortably explain after the event.

When the relevant facts live in separate trackers, Git records, CI logs, deployment systems, and incident tickets, reconstruction becomes a manual integration exercise. The practical question is therefore not whether the team pays for scope discipline. It is whether the team pays before the change, through a reusable authorization that the pipeline enforces automatically at the configured gates, or after the change, through repeated review and reconstruction.

The question Chapter 1 leaves open

What would software delivery look like if the boundary were approved before code existed, checked outside the AI agent, and kept attached to the change through production?

CHAPTER 2

The CEED Operating Model

The CEED Model reorganizes delivery around a simple division of labor: people authorize the change, AI and engineers produce it, existing tools generate evidence, and the pipeline enforces the approved decision.

2.1 AI-assisted and AI-native delivery are different operating models

AI-assisted delivery adds AI tools to an existing workflow. Engineers remain the primary producers and use AI to write, explain, test, or review code. The underlying process is still designed around people creating the change and other people inspecting it.

AI-native delivery changes the center of gravity. AI agents produce much of the routine output, while accountable people approve intent and repository scope before implementation, decide exceptions when an agent challenges that scope, and retain judgment over material release risk. The pipeline is designed for that separation from the start.

The CEED Model is designed for the AI-native direction, but a team does not need to wait until AI writes most of its code. It can introduce the same authorization, enforcement, and recording structure now and keep it as the production mix changes. The point is not to label a team as advanced. It is to ensure that more automated production does not mean less explicit accountability.

2.2 The CEED Framework adds two major contributions to the existing delivery process

The first contribution is per-change authorization that the delivery pipeline enforces automatically once the gates are wired. A Delivery Contract turns the approved requirement, allowed and protected path scope, evidence obligations, risk tier, and release rule into structured input for the delivery pipeline. The CEED Router compares the exact candidate and release evidence with that approved record, then returns the decision that branch protection or the deployment workflow requires.

The second contribution is standardization. A private script can enforce path scope, and a vendor platform can create a strong integrated workflow. What those approaches do not automatically provide is a common, producer-neutral Contract and Passport that another implementation, another team, or a future toolchain can read without knowing the original custom integration.

The two contributions need each other. Enforcement without a common record ends at the local gate. A common record without enforcement becomes documentation that can be ignored. The CEED Framework joins the two: the Delivery Contract states the authorization, the Router enforces it at the configured gates, and the Change Passport preserves the decisions and outcomes in one change history.

2.3 The delivery loop has five moments

The CEED Model uses five moments to make the lifecycle easy to follow. The table also shows who owns the decision and where the result lives, so the responsibility split does not have to be reconstructed from separate lists.

Moment	What happens	Primary owner	Where it is defined or recorded
1. Authorize	The team records the approved goal, allowed and protected paths, required evidence, risk tier, and release rule in the Delivery Contract before implementation begins.	Requirement owner, engineering lead, and relevant specialists	Requirement system, repository policy, issued Delivery Contract, and contract-issued Passport entry
2. Produce	A developer, an AI coding agent, or both create the candidate. The Contract may guide the work, but it does not give the agent authority to alter the boundary.	Developer or AI coding agent	Working branch and exact Git candidate
3. Check merge	The Router reads the candidate's changed paths and candidate-bound evidence, compares them with the approved Contract, and records approved, review-required, or blocked.	CEED Router, Git platform, CI, and evidence providers	Merge-check entry in the Change Passport
4. Check rollout	The deployment platform executes the approved strategy. At each required checkpoint, the Router verifies the strategy, sequence, and checkpoint-bound evidence before the rollout may continue.	Deployment platform, CEED Router, and release owner for exceptions	Deployment-check entries in the Change Passport
5. Learn	After the observation window, production status is recorded. Later incident associations, retractions, or a corrective cycle can be appended without rewriting earlier entries.	Observability and incident systems, authorized human owners, and Passport writer	Production entries in the Change Passport; detailed analysis remains in the incident system

A merge queue may add an earlier batch-admission check so incompatible changes are separated before a combined build. That is an integration point within the merge path, not a sixth moment in the model.

2.4 People grant authority; machines apply it

The responsibility split above is not a statement that every decision must wait for a meeting. Repository policy can allow deterministic checks to approve routine candidates automatically when the approved conditions are satisfied. What it does not allow is the producer silently changing those conditions.

A person or AI system may draft a Delivery Contract. Security, QA, architecture, and domain specialists can define evidence baselines in repository policy and review change-specific additions when their risk area is involved. The engineering lead approves the per-change path boundary, risk tier, and any later scope change through the protected Contract issuance process. The Router applies those approved rules and records the result in the Change Passport.

The automation boundary

AI may draft, analyze, warn, and recommend. It cannot issue, approve, or widen its own authorization. Machines apply approved deterministic rules automatically; accountable people remain responsible for creating or changing the authority behind those rules.

2.5 One running example makes the model concrete

To make the model easier to follow, the rest of the paper uses one small change request, REQ-1432, as a running example. It turns the five moments and the responsibility split into a practical journey. REQ-1432 shows the distinction the CEED Model is designed to make: whether the code works and whether the change was authorized are separate questions.

A support team wants its ticket-management API to store a priority of low, medium, or high so urgent tickets surface first. The approved requirement lives in the team's existing tracker as REQ-1432; the tracker remains the source for the full requirement and acceptance criteria.

Before implementation, the first Delivery Contract allows changes to the API schema, a migration directory, selected service files, and tests. It keeps the authentication and billing trees protected. The AI coding agent then edits `src/billing/config.yaml` because the feature needs a billing flag. The code works and every required test passes. The Router still blocks the merge because that file was outside the authorization in force for the candidate. The block is the point: passing tests did not authorize the billing-file change.

In this example, the engineering lead reviews the block and decides that the exact billing configuration file is genuinely required. Version 2 of the Contract authorizes that file and adds a billing integration test. The next candidate is checked against the new authorization. The earlier block remains correct because the later decision changes what a later candidate may do; it does not rewrite what version 1 allowed.

The Change Passport—the append-only history for this one governed change—keeps both decisions and the later rollout and production records under one change identity. Chapter 3 returns to REQ-1432 to unpack its Delivery Contract; Chapter 4 follows the same change through the Passport. The planned Version 1.0 CEED in Practice companion will carry the complete stored journey, including the Contract files, Passport entries, rollout checkpoints, late incident, corrective cycle, batching cases, and local report.

CHAPTER 3

The CEED Delivery Contract

The Delivery Contract turns an approved requirement into a compact, machine-readable authorization before implementation begins.

3.1 Seven fields carry the decision the pipeline needs

Each change has one canonical Delivery Contract file. Every approved version contains exactly seven top-level fields. The field set is deliberately small: each field becomes something people must approve and tools must produce, read, enforce, and preserve.

Field	Question it answers	REQ-1432 example
Requirement	What approved work is this change meant to perform, and where is the authoritative source?	Add priority to support-ticket records; tracker://REQ-1432
Allowed scope	Which repository paths may this change modify?	Ticket API schema, migration directory, selected services, and tests
Protected scope	Which paths are deliberately fenced off in this change's authorization context?	Authentication tree and billing tree

Field	Question it answers	REQ-1432 example
Required merge evidence	Which existing tests, scans, or validations must pass for the exact candidate?	Unit tests, migration validation, and API contract test
Risk tier	Which policy baseline governs approval, evidence, release, and observation?	standard
Release rule	Which strategy, ordered checkpoints, and checkpoint evidence are approved for this change?	Canary; initial exposure followed by full promotion
Schema version	Which versioned CEED rules define this Contract and its related records?	1.0

The requirement field references the organization's existing source of truth rather than replacing it. The Contract is the delivery authorization for that requirement: it captures the part of the decision that the pipeline must be able to evaluate deterministically.

3.2 Every changed path is authorized, protected, or outside the allowed scope

The `allowed_scope` and `protected_scope` fields create a three-way boundary. A path can be explicitly allowed, explicitly protected, or listed in neither set. An unlisted path is not free to change simply because it is not marked protected; it remains outside the authorization for this work.

The distinction matters. An out-of-allowed-scope result means the candidate left the area approved for the change. A protected-scope result means it crossed a boundary the team explicitly called out. Both block by default, but the second carries a stronger signal about what was crossed and may require a different reviewer or response.

Within one Contract, a more specific allowed path can create a narrow exception inside a broader protected tree. That is how REQ-1432 version 2 can allow one reviewed billing configuration file while the rest of billing remains protected. The Version 1.0 Path Pattern Profile and specificity rules are designed so independently built implementations classify the same path the same way. Agreement across independent implementations has not yet been demonstrated.

Repository policy sits above every Contract

A Delivery Contract cannot override protections set in repository policy. If policy locks down a path for all changes, no per-change Contract can authorize it, regardless of how specific the Contract pattern is.

The Router checks every path Git reports in the candidate diff, including renamed, deleted, generated, binary, vendored, lock, migration, and build files. It classifies the repository path. It does not inspect the internal content of an allowed file or prove that code inside that file cannot affect a protected system at runtime. Code review, testing, build isolation, security analysis, architecture controls, and runtime controls remain responsible for those questions.

3.3 The Delivery Contract references existing evidence instead of rebuilding it

The Delivery Contract does not introduce a second test system. It names evidence that existing tools already produce. Unit tests, migration checks, static analysis, dependency scans, API contract tests, and other controls stay in their current platforms; the Router reads the result and verifies that it belongs to the candidate or deployment under evaluation.

Two evidence moments must remain separate. Required merge evidence answers whether the exact code candidate may merge. Checkpoint evidence answers whether a particular rollout may advance. The second type often does not exist until the deployment has started. A canary error rate, for example, cannot be evaluated before the canary has served traffic.

Evidence is therefore bound to the thing it supports. Merge evidence names the candidate's base and head commit. Release evidence names the deployment and checkpoint for which it was produced. This prevents a passing result from an earlier candidate or a different deployment attempt from being reused as though it applied to the current decision.

3.4 The risk tier selects the baseline; the release rule freezes the approved plan

The risk tier and release rule serve different purposes. The risk tier is an opaque policy key such as low, standard, or high. Repository policy uses that key to select the approval rules, evidence baseline, release baseline, and observation window that normally apply.

The release rule is the resolved plan for this particular Contract version. At issuance, the relevant policy baseline and any approved change-specific additions are written into the Contract in full. The approver can therefore read one file and see the actual strategy, the ordered checkpoints, and the evidence required at each checkpoint. A later policy change does not silently alter an authorization that was already approved; applying the new baseline requires a new Contract version.

The Router does not infer what the word canary should mean. It treats the strategy identifier as an agreed value and checks the explicit checkpoint sequence recorded in the Contract. The deployment platform remains responsible for executing the rollout.

Step	What happens	What the Router verifies
1. Resolve at issuance	Repository policy supplies the baseline for the selected risk tier. Any approved addition is written into the Contract.	The Contract is structurally valid and contains the complete release rule before approval.
2. Start the deployment	The deployment platform builds or selects the artifact and invokes the first required checkpoint.	The artifact was built from an exact commit with an approved merge-check.
3. Verify the strategy	At the checkpoint marked for strategy verification, the deployment platform provides an authenticated statement of the strategy it is running.	The reported strategy matches the Contract value after any versioned platform vocabulary mapping.
4. Check checkpoint evidence	The platform or evidence provider supplies the evidence required for that checkpoint.	Each item belongs to the same deployment_ref and checkpoint, and all earlier checkpoints are approved.
5. Continue or hold	The deployment workflow waits for the Router response.	The Router records approved, review-required, or blocked in the Change Passport before the rollout continues.

A direct release may use one checkpoint. A canary release normally uses at least two: one to begin limited exposure and another to promote the change more widely after the required evidence exists. Production observation follows the rollout; it is not another release checkpoint.

For REQ-1432, the canary release begins at initial-exposure: the deployment is bound to the approved artifact, the platform's authenticated strategy is checked against canary, and the required smoke-test evidence must pass before limited exposure is approved. At full-promotion, the Router confirms that the first checkpoint is already approved and then checks the canary error-rate evidence before wider promotion.

3.5 Contract versions are different authorizations, not edits to one mutable file

When the allowed or protected path scope, or another approved field, changes, the team does not modify the earlier authorization in place. It issues a new immutable Contract version through a new Contract-only commit and protected approval.

In practice, the canonical path stays stable: `.ceed/contracts/REQ-1432.yaml`. Version identity comes from the contract-issued Passport entry, which records a logical identifier such as `REQ-1432@v1` or `REQ-1432@v2` together with the exact file path, immutable Git commit, and SHA-256 content digest. A later version also names the version it supersedes. Every merge-check and deployment-check entry names the Contract version it used.

The Router therefore does not guess which bytes are current. The gate invocation names the issued version; the Router verifies the corresponding contract-issued entry, checks that the issuance commit is in the candidate's ancestry where required, and confirms that the Contract bytes match the recorded digest. Previous versions remain retrievable for as long as a retained Passport refers to them, whether through Git history or an archive that preserves the referenced object.

REQ-1432 version 2 authorizes one additional billing configuration file and requires an additional integration test. The Change Passport records that version 2 supersedes version 1 and preserves the exact objects and decisions. The business reason for widening scope remains in the organization's authoritative work or review record; Version 1.0 does not add a free-text rationale field to the seven-field Contract.

Why versioning matters

The history can hold both truths. The first candidate violated the authorization that existed at the time, and a later approved Contract changed the authorization for a later candidate. The earlier block remains correct because the later version does not rewrite the authorization that governed it.

3.6 REQ-1432 as an approved Contract

The summary below uses the schema's field names while keeping the example readable. The complete YAML will be available in CEED in Practice as part of the planned Version 1.0 package.

Illustrative Contract summary

```
requirement: REQ-1432 - Add priority to support-ticket records
allowed_scope: API schema, migration directory, selected services, tests
protected_scope: authentication tree, billing tree
required_merge_evidence: unit tests, migration validation, API contract test
risk_tier: standard
release_rule: canary; initial exposure -> full promotion
schema_version: 1.0
```

The Contract does not attempt to carry the entire requirement, design, threat model, or incident procedure. It references and coordinates those sources. Its job is to state the authorization the delivery system must be able to enforce.

CHAPTER 4

The CEED Change Passport

The Delivery Contract says what the change is allowed to do. The Change Passport preserves which authorization was used, what each gate decided, and what was later learned in production.

4.1 One change has one append-only history

A CEED Change Passport is an append-only sequence for one change. Tooling writes entries in order and does not update or delete an earlier entry through the CEED interface. The record grows as the change moves through Contract issuance, merge, deployment, production observation, and later knowledge.

Append-only does not mean that the underlying storage product must be a blockchain or a particular immutable database. Version 1.0 defines the information model and write interface. An organization may add signatures, WORM storage, external anchoring, or stronger access controls according to its own security needs.

The Passport has no terminal status. A deployment may appear healthy at the end of its observation window and later be associated with an incident. The later fact is appended. A later post-mortem may retract that association, which is also appended. The earlier entries remain because they show what the organization knew and decided at each point in time.

4.2 Exact references keep the history attributable

Four identifiers prevent the record from collapsing different operational events into one.

Identifier	What it names	Why it must remain distinct
Commit SHA	One immutable Git commit and the code state it represents	A rebase, squash, force-push, or queue reconstruction creates a different candidate that needs its own decision.
Change Passport UUID	The complete history of one governed change	Every entry for the change uses the same spine even when several Contract versions or deployments occur.
deployment_ref	One deployment event	The same commit can be deployed more than once, and each rollout needs its own checkpoints, evidence, and outcome.
incident_ref	One incident in the organization's incident system	The same incident may implicate several changes; detailed causal analysis remains in the incident record.

In practice, REQ-1432 keeps Passport UUID 9f2b7c14-3e8a-4d21-b6f0-1a2c3d4e5f60 for the whole change. The first candidate, head SHA bbbbbb, is blocked; the later candidate, head SHA cccccc, is approved under version 2. The production rollout is recorded as deployment_ref DEP-5501. When incident INC-2213 is opened later, incident_ref incident://INC-2213 points that later knowledge back to DEP-5501 without changing the Passport identity.

The Passport also records exact bindings. A contract-issued entry ties a logical Contract version to its path, issuance commit, and content digest. A merge-check entry ties that version to the candidate's base and head commit. A deployment-check entry ties the Contract version and artifact digest to one deployment_ref and checkpoint. A production entry ties the observed outcome to that same deployment_ref.

These links make the history navigable without pretending that the Passport replaces every source system. The requirement, full CI logs, deployment details, and incident analysis can remain in the systems that own them; the Passport preserves the references and decisions needed to understand how they relate to the authorized change.

Version 1.0 provides cross-producer lifecycle traceability through one governing authorization and one Change Passport identity. Facts from different tools or producers are bound to the same Contract version, candidate, deployment, and later production history. It does not standardize exact per-agent, per-action, per-file, or per-line provenance. Finer producer attribution is later work.

4.3 The running example as a Passport timeline

The table shows the core entries from REQ-1432 in abbreviated form. It is not a substitute for the schema, but it makes clear what is actually recorded.

Entry	Key fields captured	What the entry establishes
Contract v1 issued	contract_version REQ-1432@v1; path; issuance commit; content digest; issuer; approver; policy version; risk tier	The exact first authorization existed before the candidate was evaluated.
First merge-check blocked	Contract v1; candidate base/head; protected billing path; passing evidence; status blocked	The tests passed, but the candidate crossed the authorization in force.
Contract v2 issued	contract_version REQ-1432@v2; exact new commit and digest; supersedes REQ-1432@v1	A new authorization was approved without changing the earlier one.
Merge-check approved	Contract v2; exact candidate; evidence items bound to its base/head; status approved	This candidate satisfied the second authorization at the merge gate.
Initial-exposure checkpoint	deployment_ref DEP-5501; artifact digest and built_from_sha; strategy attestation; smoke-test evidence; status approved	The limited rollout used the approved strategy and was approved to begin.
Full-promotion checkpoint	same deployment_ref and artifact; canary evidence; status approved	The earlier checkpoint was complete and the rollout was approved to advance.
Production observation	same deployment_ref; status healthy; outcome_source; rolled_back false; checkpoint completeness complete	The observation window closed with a sourced, attributable result.
Late incident	same deployment_ref; incident_ref; incident_opened_at; reason late-incident	New knowledge was attached to the original rollout without inventing a second deployment.
Corrective cycle	Contract v3 and a new candidate, deployment_ref, checkpoints, and production outcome	A fix made solely to restore the original change can continue in the same Passport history.

A corrective cycle stays in the original Passport only when its sole purpose is to restore the intended outcome of that change. A fix with independent intent, or one that spans several original changes, receives its own Delivery Contract and Change Passport. Relationships among those changes remain in the organization's work and incident systems.

4.4 Production entries show not only the outcome, but what stands behind it

At the end of the policy-defined observation window, the implementation records healthy, incident, or inconclusive. The status alone is not enough. The entry also captures how the result arrived, whether the deployment was rolled back, and whether the required checkpoint sequence was complete.

Field	What is recorded	Why it matters
outcome_source	automated or manual mode; provider; authenticated subject; retrievable reference; and hash of the exact payload consumed	A monitoring system reporting healthy and a person typing healthy are visibly different claims rather than indistinguishable text.
rolled_back	true or false	Rollback is an action, not an outcome category. A deployment can be incident and rolled back, incident without rollback, or rolled back for precautionary reasons.
checkpoint_completeness	complete, incomplete, or unknown	Complete means every required checkpoint has an approved entry for the deployment. Incomplete means at least one did not. Unknown covers the real case where the Router cannot establish either answer.
incident_ref and incident_opened_at	The authoritative incident identifier and its opened timestamp when the outcome is incident or a late incident is associated	Several Passports can point to the same incident without claiming that any one change is the sole cause.

If a later analysis clears an incident association, a compensating retraction entry uses the same deployment_ref and incident_ref. Reporting reads the latest appended state for that pair, but the original association remains visible. The record therefore corrects the conclusion without pretending the earlier conclusion never existed.

4.5 The common shape is what makes the history portable

A private record can work well until the people and tools that understand it are no longer present. An auditor may ask what was approved eighteen months ago. A new team may take over the service. The organization may replace its coding agent, CI platform, policy engine, or deployment system.

Producer-neutral fields make the authorization history survive those changes. The underlying CI log or incident report can remain authoritative in its original system, while the Delivery Contract and Change Passport preserve the standard identifiers, references, and decisions that explain how the evidence related to the change.

Portability is therefore not a claim that every tool must store the same raw data. It is a claim that independently built tools should be able to understand the same authorization and lifecycle record without relying on a proprietary join known only to the original implementation.

CHAPTER 5

The CEED Router

The CEED Router is the enforcement and orchestration component. It turns the approved Contract into gate decisions and writes those decisions to the Change Passport.

5.1 What the Router reads, checks, writes, and leaves to other systems

The Router's role is easier to understand when its boundary is explicit.

Action	What the Router does
Reads	The issued Contract reference; the exact changed-file list from Git; candidate-bound merge evidence; artifact and deployment information; strategy assertions; checkpoint-bound release evidence; and production facts supplied by configured providers.
Checks	Contract validity and issuance; path authorization for every changed path; evidence presence, result, and binding; exact-candidate approval; batch attribution where applicable; release-strategy equality; checkpoint order; and deployment evidence.
Writes	A contract-issued, batch-admission, merge-check, deployment-check, or production entry to the Change Passport before returning the associated decision where a gate is involved.
Leaves to other systems	Requirement approval, code generation, code review, test execution, artifact building, rollout execution, observability, incident analysis, and the internal integrity of the platforms that supply facts.

The Router is not an AI code reviewer. At merge it asks whether each path Git reports is authorized, protected, or outside the allowed scope. It does not judge the logic inside an allowed file. A candidate can be within scope and still fail a security scan or human review; it can also pass every test and still be blocked because it touched a protected path.

5.2 The merge decision applies to one exact candidate

A candidate is identified by its base and head commit SHAs. The Router validates the applicable Contract, reads the changed-file list for that exact pair, classifies every path, and checks the evidence named by the Contract. Each evidence item must refer to the same candidate, not merely to a branch name or a previous head commit.

The Router records all failed rules before returning approved, review-required, or blocked. Branch protection treats that result as a required status check. A later rebase, force-push, base change, squash, or queue reconstruction produces a different candidate and therefore requires another decision.

The same principle applies to the commit that actually ships. If the final commit exists before merge, a queue can evaluate it prospectively. If a squash or rebase creates a new commit only when it lands, the deployment gate holds the artifact until that final SHA has its own approved merge-check. Being on the default branch is not the same as being approved for deployment.

5.3 The Router checks rollout checkpoints; it does not perform the rollout

At deployment, the platform assigns or receives a `deployment_ref` and presents the artifact, built-from commit, Contract version, and current checkpoint. The Router confirms that every constituent change in the artifact has the required exact-commit approval, that the approved release strategies are compatible, and that the checkpoint is next in the ordered sequence.

At the checkpoint marked for strategy verification, the deployment platform supplies an authenticated statement of the strategy it is actually running. The Router compares that value with the Contract. It then checks the evidence required for the current checkpoint and confirms that each item belongs to this deployment and this checkpoint. Missing, mismatched, or out-of-order information blocks by default.

The deployment platform executes the canary, direct release, progressive rollout, or feature exposure. The Router supplies the decision the workflow must wait for before it advances. After the governed rollout reaches production, the policy-defined observation window produces a production entry. That observation is recorded separately; it is not another release checkpoint or release-gate decision.

One sequence, three different responsibilities

The deployment platform performs the rollout. Evidence providers report what happened at a checkpoint. The CEED Router decides whether the approved rule has been satisfied and records that decision in the Change Passport.

5.4 Batch admission can catch incompatibility before a combined build

When a merge queue combines several separately authorized changes, their Contracts do not merge. Each constituent remains governed by its own Contract and Passport. The integration must attribute a complete, non-overlapping set of changed paths to every constituent.

Where the platform exposes pending candidates before constructing a group, the Router can check path overlap and release-rule compatibility at admission. Incompatible changes can wait for separate groups before a combined build begins. Where no pre-construction hook exists, the same check occurs against the prospective or formed group, and the queue may need to discard and rebuild it. The CEED Framework makes that limitation explicit rather than assuming every queue has the same topology.

A final overlap check remains necessary because candidates can change after admission. Generated files may appear, a rebase may change the diff, or a platform may lack an early hook. The merge-gate floor checks the constituents as they actually exist before merge.

5.5 Fail-closed behavior depends on a real gate and trusted inputs

Where a branch rule or deployment workflow invokes the Router and waits for its response, the integration fails closed. An invalid or unsupported Contract, missing evidence, a strategy mismatch, an unavailable decision, or a failure to write the required Passport entry does not become an approval.

That guarantee applies only to governed paths that are actually wired to the Router. A deployment workflow that never invokes the checkpoint, or invokes it without waiting, cannot be stopped by CEED. Passport-native reporting identifies malformed or incomplete sequences among the Passports it reads, but it does not discover activity for which no Passport exists.

The Router also relies on authenticated metadata from Git, CI, deployment, observability, and incident systems. It does not independently prove the internal integrity of every provider or administrator. Existing

platform security, access control, audit logs, runner hardening, and separation of duties remain part of the trust boundary. A bypassed action must not be represented as CEED-approved.

5.6 Local reporting uses the same record after the gate decision

The CEED Router includes a local reporting capability. It reads a consistent snapshot of local Change Passports, applies the Version 1.0 calculations defined in the current working package, and produces an inspectable report file and digest. Report generation remains outside the merge and deployment critical path and performs no network access.

This gives teams one maintained implementation of the initial measures instead of requiring every adopter to invent its own formulas. The output can be reviewed locally or loaded into existing analytics tools. The reporting boundary and optional cross-organization comparison are explained in Chapter 9.

CHAPTER 6

What a Team Sets Up

A CEED implementation adds a small control layer around the delivery path. The existing systems keep their specialist roles; the new work is to give them one authorization to follow and one history to write.

6.1 Existing systems remain authoritative for their own facts

The CEED Framework is not a replacement tool suite. It coordinates systems that already know different parts of the delivery story.

Existing system or role	What remains authoritative there	What the CEED Framework adds
Requirement or work-management system	The approved requirement, acceptance criteria, discussion, and planning context	A stable reference to the requirement in the Delivery Contract
Git and repository platform	Commits, changed paths, pull requests, protected approvals, branch rules, and merge queues	Protected Contract issuance, exact-candidate input, and required Router decisions
CI, test, and security providers	Tests, scans, analysis, and the evidence payloads they produce	Named evidence obligations and candidate or deployment binding
Deployment or feature-management platform	Artifact creation, rollout execution, exposure control, and platform-native deployment history	A resolved per-change release rule and checkpoint decisions the workflow must wait for
Observability and incident systems	Production signals, incidents, and detailed root-cause analysis	Attributable outcome and incident references in the Change Passport
Human owners	Intent, architecture, risk, exception, and release judgment	An explicit place to approve the per-change authorization and review exceptions
CEED Router and Passport store	CEED decisions and lifecycle sequence	Deterministic comparison, append-only recording, and local reporting

This separation avoids two common misunderstandings. The CEED Framework does not copy every source record into the Passport, and it does not ask the Router to become the system of record for work

management, testing, deployment, or incidents. It preserves the decision and the reference needed to connect those facts to the authorized change.

6.2 The repository gains policy and one Contract file per change

A repository adopts a protected `.ceed` directory. Repository policy sits beside a contracts directory containing one canonical Delivery Contract path for each change. Application source and test directories remain where they are.

Illustrative repository layout

```
repo/
  .ceed/
    policy.yaml
    contracts/
      REQ-1432.yaml
      REQ-1440.yaml
  src/      # existing source
  tests/    # existing tests
```

Repository policy defines the controls that are reused across changes: protections no Contract may override, approval rules by risk tier, evidence and release baselines, observation windows, provider mappings, and authorities for manual outcomes or incident updates. The contracts directory and policy file are protected by the Git platform.

Each Contract version is proposed through a Contract-only issuance commit. The validator checks the seven-field shape, path rules, selected risk tier, and resolved evidence and release requirements. The protected approval is tied to that exact commit, after which the Passport writer records the issued version. Code generation can then proceed from the approved authorization.

6.3 Setup work and per-change work should not be confused

The largest integration tasks are repository or platform work performed once and then reused. The per-change work is the decision that must remain specific to one change.

Cadence	Work	What it replaces or organizes
Once per repository or delivery path	Protect <code>.ceed</code> ; define repository policy; connect Git, CI, deployment, outcome, and Passport-store interfaces; wire required gates; configure local reporting	Existing approval rules, evidence sources, release practices, and platform integrations that were previously spread across tools or custom scripts
When policy changes	Update risk-tier baselines, provider mappings, observation windows, and approval authorities through the repository's protected policy process	Repeated interpretation of team-wide rules inside each ticket or release discussion
Once per Contract version	Draft, review, validate, and issue the seven-field authorization	Fragments of the same boundary now repeated across a ticket, prompt, pull-request description, and reviewer explanation
For each candidate	Produce code and evidence; let the Router evaluate the exact diff and evidence	Manual re-discovery of whether every changed path belonged to the approved work
For each deployment	Invoke the required checkpoints; record the production outcome and later knowledge	Disconnected release checks and later reconstruction of which authorization governed the rollout
Periodically	Generate the local Passport-native report	Private formulas or one-off manual calculations from change records

The table does not mean every organization performs all of today's work well or consistently. It shows the intended shift: reusable policy is configured once, the per-change authorization is approved once per version, and repeatable comparisons are delegated to the pipeline.

6.4 A team can phase implementation without overstating conformance

A team may introduce the components in stages. It can begin by producing valid Contracts and Passports, then wire merge enforcement, deployment checkpoints, and local reporting as its delivery path becomes ready. Component-level progress can be useful before the complete operating model is in place.

The conformance claim must match the implementation. A Contract producer can conform as a Contract producer. A Passport writer and store can conform to the Passport writer and store class. A gate implementation or reporter can be tested separately. A full delivery implementation requires all four classes and requires merge and deployment systems to wait for the applicable decisions. Chapter 9 explains what the Conformance Pack tests and what it does not certify.

6.5 The complete path is simple enough to draw

What a platform lead should be able to draw

Approved requirement -> issued Delivery Contract -> developer or AI-generated candidate -> Router merge decision -> deployment platform and Router checkpoints -> production outcome -> local report.

Each arrow uses a system the organization already runs. The CEED Framework supplies the common authorization, decision vocabulary, and append-only history that keep the arrows connected.

The architecture is deliberately modular. A team can replace a coding agent, policy engine, CI provider, deployment platform, or reporting interface without changing the meaning of the Contract and Passport, provided the replacement continues to honor the Version 1.0 semantics.

CHAPTER 7

Existing Tools and the Missing Common Language

Modern delivery systems already contain most of the mechanisms the CEED Framework needs. The contribution is a producer-neutral way to make those mechanisms act on the same approved change.

7.1 The component controls already exist

Requirements tools can structure work before implementation. Git platforms compute diffs, protect branches, and route approvals. Policy engines evaluate structured rules. CI and security systems produce evidence. Attestation formats can bind claims to artifacts. Deployment and feature-management platforms control exposure. Observability and incident systems record what happened in production.

This matters because the CEED Framework does not depend on a new computing primitive. Reading a diff, validating structured data, comparing paths with policy, checking evidence, holding a deployment checkpoint, appending an entry, and calculating a report are established operations. The CEED-specific work is to make all of them refer to the same per-change authorization and history.

7.2 Adjacent tools solve important parts of the problem

The companion Tool Verification Matrix is a dated primary-source review rather than a permanent market claim. It asks whether tools begin from an approved requirement, distinguish allowed from protected scope, represent evidence obligations, scale controls by risk, enforce a gate, and connect production outcomes to the governing change. Products and standards can be rechecked as they evolve without silently changing the historical evidence record.

Artifact or capability	What it contributes	What the CEED Framework adds or standardizes
Agent instructions, repository rules, and IDE context	Guide the agent, provide repository knowledge, and discourage unwanted actions while work is being produced	An approved per-change authorization enforced outside the agent against the exact candidate
Requirement-first and specification-driven workflows	Turn intent into requirements, plans, specifications, and tasks before implementation	A compact path, evidence, risk, and release authorization that the configured delivery gates enforce automatically
CODEOWNERS, rulesets, and branch protection	Protect repository areas, require reviews, and prevent merge until configured checks pass	A dynamic per-change allowed/protected boundary and a lifecycle record tied to the specific authorization
OPA and other policy engines	Evaluate structured input against programmable rules	The CEED domain objects, path semantics, decision vocabulary, and lifecycle behavior those engines may implement
in-toto, SLSA, DSSE, and Sigstore	Carry provenance, predicates, subjects, attestations, and signatures	The authorization-to-outcome semantics that may be represented or protected using those substrates
Testing, analysis, and security tools	Produce evidence about behavior, quality, dependencies, and risk	A prior declaration of which evidence is required and binding to the exact candidate or checkpoint
Deployment and feature platforms	Execute direct, canary, progressive, or controlled exposure	A resolved per-change release rule and ordered decisions recorded against the same Change Passport
Observability and incident systems	Provide production facts and detailed incident analysis	References that attach later knowledge to the original authorized change without replacing the source system

The important comparison is not whether the CEED Framework can replace these tools. It should not. The question is whether their outputs share a common, portable statement of what one change was authorized to do and a common history of the decisions and outcomes that followed.

7.3 Producer neutrality is the architectural choice

A vendor-native solution can provide strong controls and a simpler integrated experience. The tradeoff is that the authorization and history may be expressed in a format only that vendor fully understands. That may be acceptable for an organization that values one platform more than portability.

The CEED Framework deliberately chooses the other side of the tradeoff. The Contract and Passport use versioned shapes defined for the intended Version 1.0 release. A policy engine can evaluate them, an attestation format can carry or sign selected statements, a deployment platform can enforce checkpoint

decisions, and an analytics product can consume the report output. None of those choices changes the meaning of the underlying CEED record.

The CEED Framework does not depend on vendors leaving CEED-compatible semantics outside their products. Native support for those semantics is a successful outcome for an open, producer-neutral standard. The control boundary remains the same: the producer whose output is governed must not be able to grant, widen, reinterpret, or bypass its own authority. The durable value lies in canonical, versioned semantics, conformance with those semantics, portability across toolchains, and actual use in delivery practice.

The CEED Framework standardizes operational semantics rather than dictating a single technology stack.

A conforming implementation can use the tools an organization already trusts. Toolchain neutrality comes from keeping the authorization, identifiers, decisions, and reporting meaning stable even when the producers change.

This is also where the value flows in both directions. The CEED Framework gives coding-agent and delivery platforms a standard authorization object they can read, draft, display, warn against, or enforce. Those platforms give CEED richer repository context, better developer experience, stronger evidence, and native gate integration. Product-specific examples should be added only when the dated Matrix verifies their current capabilities and boundaries.

7.4 The missing piece is portable authorization-to-outcome semantics

Looking across the adjacent capabilities shows a repeated boundary:

- A requirement-first tool can organize what should be built without defining a protected repository boundary that an external gate enforces.
- A path-control tool can protect code ownership or evaluate policy without showing which approved requirement authorized this specific change.
- A supply-chain attestation carries a verifiable claim about how an artifact was built, but not the business intent and later production outcome.
- A deployment platform can control exposure without preserving the complete earlier authorization that led to the release.

An organization can connect these systems through private identifiers and custom scripts. That can work, but the result remains a private integration. Another implementation cannot assume the same fields, another team cannot calculate the same measures without translation, and an auditor may still need the original systems or the people who understand the custom joins.

Why use a producer-neutral standard

A common model lets different tools produce and consume the same authorization history, lets organizations calculate the same defined measures, and gives audit and governance readers a stable record that does not disappear when the toolchain changes.

7.5 Standardization must be testable, not only well written

A schema is useful, but interoperability depends on behavior as well as field names. Two implementations must classify the same path, bind the same candidate and evidence, enforce the same checkpoint order, preserve the same append semantics, and calculate the same report from the same input.

The intended Version 1.0 package therefore includes the Conformance Pack. It provides positive and negative examples, semantic checks, 80 behavioral vectors, five report fixtures, and expected outputs. An implementation can run those cases and show whether its decisions agree with the current model.

Passing the pack does not prove production security, availability, latency, recovery, or suitability for every repository. It establishes something narrower and necessary: agreement with the cases the standard has actually defined. A separately written implementation that reaches the same results without importing the reference runner remains the strongest next interoperability test.

7.6 What the current evidence establishes

Sections 7.1 and 7.5 establish the two boundaries that matter here: the required operations already exist, and agreement with the current conformance cases is not the same as field validation. The table below states what the current evidence supports.

Claim	Current basis	Status
Implementable with existing delivery systems	Current architecture, schemas, deterministic rules, examples, and conformance vectors	Strong engineering plausibility; executable for current included cases
Independent implementations produce identical decisions	Requires separately written implementations evaluated against the same Conformance Pack	Not yet established
Reliable as a production-blocking control	Requires observed integration, failure, concurrency, security, latency, and recovery evidence	Not yet established by the current evidence
Supports a more structured audit trail	One per-change authorization and Passport history with standard bindings, references, and append semantics	Structural basis defined; effect on audit effort remains to be measured
Reduces incidents, cost, or cycle time	Requires field observation or controlled experiments	Not yet established

CHAPTER 8

What Adoption Adds, Moves, and May Reduce

Adoption does require setup and approval. The fair comparison is not CEED work versus no work; it is one explicit authorization that the pipeline checks automatically versus the same scope, evidence, and release decisions scattered across tickets, prompts, reviews, and delivery tools.

8.1 The Contract already exists in fragments

Every serious change involves decisions about what is being built, which folders and files may change, what must stay protected, which checks must pass, how much risk the change carries, and how it may reach production. Not every team makes each decision explicitly, and the quality of those decisions varies. The need for them does not disappear because they are informal.

AI users often write fragments of that decision into a prompt: change this service, do not touch billing, preserve the public API, run these tests. The next prompt repeats some of it, perhaps differently. A pull-request description repeats it again, and the reviewer compares the generated diff with the ticket and with what they remember the change was supposed to touch.

The Delivery Contract puts those fragments into one approved record. It states the repository paths the change may modify, the paths it must not modify, the checks that must pass, the risk tier, and the release rule. The same record is then available to the agent, the merge and deployment gates, the Change Passport, and the reporter.

One decision, used repeatedly

The team records the authorization once per Contract version. The pipeline then applies that same decision to the exact candidate and release checkpoints instead of asking each later reviewer to reconstruct it from prompts and diffs.

8.2 What is genuinely new

Adoption adds real platform work. A team must protect the .ceed control files, define repository policy, connect Git and evidence providers, wire merge and deployment gates, operate a Passport store, and configure local reporting. Those activities should not be minimized or described as free.

Most of that setup is reused across changes. Repository policy holds approval rules, risk-tier baselines, required evidence, release baselines, observation windows, and provider mappings. Once the gates are wired, the Router applies those rules automatically to each candidate. The per-change work is to prepare and approve the Contract version and decide any exception that changes it.

This changes what a reviewer has to discover. Without a Delivery Contract, the reviewer may have to read the ticket, prompt, pull request, and diff to work out which folders and files were supposed to change. With the Contract, the approved file scope and protected paths are written down before the code is produced, and the merge gate compares the Git diff with them automatically. Code review still examines the implementation; an out-of-scope file becomes an explicit authorization decision rather than a hidden question inside the review.

8.3 The main adoption frictions already exist in another form

Friction	How teams handle it without CEED	What the CEED Framework changes	What still requires human judgment
Preparing and approving a Contract before code can feel like administrative work	Requirement, file scope, tests, risk, and release expectations are distributed across tickets, prompts, planning notes, and pull requests	A compact Contract records those decisions once per version so the agent, gates, Passport, and reporter use the same authorization	Whether the requirement, file scope, risk tier, evidence, and release rule are appropriate
An agent needs a shared utility, lock file, or protected configuration that was not anticipated	A reviewer notices the extra file in the diff, asks why it changed, and decides whether to accept, revert, or redesign it	The Router blocks the candidate when the diff includes a file outside the approved scope or inside protected scope, and a new Contract version is required if the team genuinely wants to widen the authorization	Whether widening scope is necessary and what additional evidence or review it requires

Friction	How teams handle it without CEED	What the CEED Framework changes	What still requires human judgment
Several changes conflict in a merge queue	The queue, CI build, or reviewers discover file overlap or incompatible release assumptions after grouping has begun	An admission check can separate incompatible changes before a combined build where the platform exposes that hook; a final check still protects the formed group	How to requeue, reorder, split, or redesign the changes when a real conflict exists
A later audit or incident review needs the original decision	People join tracker history, pull requests, CI logs, deployment records, and incident tickets	The Passport preserves the exact Contract, gate decisions, evidence references, rollout checkpoints, and later knowledge in one sequence	How long to retain the sources and how much additional assurance the organization requires

The CEED Framework does not eliminate these decisions. It moves them to the point where they can still change the approved file scope, required evidence, or release path, instead of leaving them to surface only after more work has accumulated.

8.4 Scope discipline is paid for either way

A team can record and approve which folders and files a change may modify before AI produces the code, or it can ask each reviewer to reconstruct that scope from the ticket and generated diff. It can record the authorization as the decision is made, or reconstruct it later from several systems. Where a merge queue exposes an early hook, it can check file overlap before a combined build; otherwise the conflict is discovered later in the queue.

Scope discipline is paid for either way

The CEED Framework moves the reusable decision to the start of the change, before the code is written: record the approved file scope, required evidence, risk tier, and release rule once, then let the pipeline apply the same decision at merge and deployment. If the work needs something outside that authorization, the team changes the Contract before a later candidate is approved.

This is not an argument against code review or human control. It is an argument for taking one repeatable comparison out of the reviewer's head. Review should focus on whether the implementation is good, not on reconstructing whether each changed file was part of the approved work.

8.5 The Router adds comparisons, not replacement builds

The path check compares the Git changed-file list with the Contract's allowed and protected paths. The evidence check reads the tests and scans the team already runs and verifies that the required results belong to the candidate. The release check compares the deployment platform's strategy and checkpoint evidence with the Contract's resolved release rule. The Router is checking existing facts against the approved authorization, not running a second build, test suite, or rollout.

That does not establish production latency or scale. Version 1.0 has no field measurements showing how long an implementation takes in a large monorepo, how it behaves under concurrency, or whether the total cycle becomes faster. The largest human delay may be waiting for a Contract approval or scope exception, especially when the original file scope was incomplete.

The tradeoff is explicit: state the approved file scope, evidence, risk tier, and release path before implementation, and change that authorization when the work genuinely needs more, rather than discovering and reconstructing the decision after an unauthorized candidate has moved farther through delivery. Whether that reduces total effort in a particular organization must be measured in practice.

8.6 Tooling can draft the Contract without becoming the authority

Contract preparation does not have to begin with a person writing YAML from a blank file. An AI-based authoring tool could read the approved requirement, repository structure, and policy baseline and propose the seven-field Delivery Contract: requirement reference, allowed file scope, protected paths, required evidence, risk tier, release rule, and schema version.

The proposal has no authorization effect. The human approver or reviewer group required by repository policy for that risk tier reviews the proposed file scope, evidence, risk tier, and release rule; policy may also require the approver to differ from the issuer. Protected approval gives the Contract its authority. Drafting remains assistance, not authorization.

The same approved Contract could then be exposed to the coding agent through a CLI, IDE integration, or MCP server so the agent sees the allowed and protected paths before it edits them. That guidance could reduce avoidable out-of-scope edits, but it is not the control. The external pipeline gate remains decisive and checks the exact candidate independently.

One record, two uses

An authoring or agent-side tool may use the Delivery Contract to help prepare and guide the work. The pipeline uses the approved Contract as the governing authorization. Assistance can move feedback earlier; the external gate still makes the delivery decision.

The current Version 1.0 working package permits this drafting pattern but does not include a Contract-authoring tool or agent-side awareness integration.

8.7 Where the model is likely to provide the most value

The case for the CEED Framework becomes stronger as the share of AI-produced code rises, the cost of an unauthorized file change increases, more tools contribute facts across the lifecycle, or the organization must show exactly what was approved. Platform teams, regulated products, shared services, large repositories, and multi-agent delivery paths are natural candidates because the authorization history matters beyond one developer and one release.

A small team with one repository, a simple toolchain, little AI-generated work, and no outside assurance requirement may reasonably choose a lighter approach. The question is not whether the team is technically capable of running CEED. It is whether a shared authorization and portable history provide enough value for its environment.

Adoption can begin with one team and one repository. The unit of value remains one change: record which files it may modify and which are protected, state the evidence and release rule, let the pipeline enforce that authorization, and keep the history.

CHAPTER 9

Assurance, Conformance, and Learning

The CEED Framework strengthens the structure around a software change, but it does not replace quality engineering, establish legal compliance, or prove production benefit by publication alone.

9.1 The CEED Framework works with existing quality and security controls

Path authorization answers one question: did the exact candidate stay inside the repository boundary approved for this change? It does not answer whether the implementation inside an allowed file is correct, secure, maintainable, or architecturally sound.

Code review continues to evaluate design and maintainability. Tests verify expected behavior. Static analysis, dependency tools, and security scanners identify technical risks. Architecture and domain reviewers apply context. Deployment platforms manage exposure. Observability and incident systems establish what happened in production.

The CEED Framework adds the connective layer. The Contract states which evidence is required and what release path was approved. The Router checks the evidence and boundary at the relevant gates. The Passport preserves the references and decisions so the organization can see how the specialist controls related to the authorized change.

The CEED Framework still faithfully enforces a poor authorization. An overbroad Contract, an inappropriately low risk tier, or weak evidence can all be approved and then enforced exactly as written. Contract quality remains a human and organizational responsibility. What the framework changes is that those choices become explicit, durable, and available for measurement instead of remaining hidden in scattered instructions, defaults, or reviewer assumptions.

Question	Primary control
Was the changed repository path authorized for this change?	Delivery Contract and CEED Router path check
Is the code inside an allowed file correct and maintainable?	Code review, tests, architecture review, and engineering standards
Is the artifact trustworthy and bound to the evaluated source?	Build controls, provenance or attestation mechanisms, and CEED artifact binding
May the rollout advance at this checkpoint?	Deployment platform, release evidence, and CEED Router decision
What happened in production?	Observability and incident systems, referenced by the Change Passport

9.2 The record can support governance and EU AI Act work, but it is not a compliance certificate

For organizations building or changing AI systems, the EU AI Act places emphasis on risk management, technical documentation, logging, human oversight, and post-market monitoring for systems within its scope. [4] Software-delivery evidence is only one part of that broader governance picture.

The CEED record can support that work by showing how a particular software change was authorized, which human approvals and evidence applied, which artifact and release path were used, and what was later recorded in production. The append-only history can supplement an organization's change-control, assurance, and technical-documentation processes, especially when teams need to show that AI-generated code did not move through delivery on the strength of the agent's own judgment alone.

The boundary is equally important. The CEED Framework does not determine whether an AI system is high-risk, create the required risk-management system, assess data governance or fundamental-rights impacts, validate model accuracy, satisfy transparency duties, or replace the operational logs required for the AI system itself. A Change Passport is a software-delivery history, not the complete legal or technical file for an AI system.

Evidence boundary

The CEED record may support an organization's governance and documentation processes. CEED conformance does not by itself establish compliance with the EU AI Act, another law, a security standard, or an internal control framework.

9.3 A common evidence language for audit and assurance

The Delivery Contract and Change Passport give audit and assurance teams a common evidence language across clients, business units, repositories, and toolchains. Source systems remain authoritative, while the common record gives recurring questions a consistent shape: what was authorized, which candidate was checked, which evidence supported the decision, how the release advanced, and what was later recorded in production.

That common shape creates a common audit grammar. Reusable procedures can check whether the required records exist, whether Contract and candidate bindings match, whether release checkpoints were completed in order, and whether later corrections were appended without rewriting earlier decisions. The procedure can remain stable even when the underlying Git, CI, deployment, or incident platforms differ.

The CEED Framework standardizes the evidence trail, not the auditor's conclusion. It does not decide whether a control was sufficient, an exception was justified, or an organization complied with a law, standard, or internal policy. Whether this common trail reduces audit effort or cost, and by how much, remains a field-evidence question.

9.4 Conformance asks whether implementations agree

A standard becomes useful when independently built tools can produce and honor the same records. Version 1.0 therefore defines component conformance classes rather than one vague claim.

Conformance class	What a passing implementation demonstrates
Contract producer	Produces a valid seven-field Contract, resolves the release rule at issuance, and preserves every issued version
Passport writer and store	Writes valid entries in append order, handles retries idempotently, and provides consistent reads

Conformance class	What a passing implementation demonstrates
Gate implementation	Evaluates the exact candidate and deployment against the applicable Contract and records the decision before allowing the governed action
Passport-native reporter	Calculates the Version 1.0 measures deterministically from local Passports, creates an inspectable output and digest, and makes no network call during generation
Full delivery implementation	Satisfies all four classes and wires merge and deployment systems to wait for the applicable decisions

The Conformance Pack supplies the schemas, examples, semantic checks, behavioral vectors, report fixtures, and golden outputs used to test those claims. A passing result shows agreement with the current packaged cases. It does not certify security, production readiness, availability, performance, or an organization's governance quality.

Conformance also protects producer neutrality. An implementation can use a different policy engine, storage system, Git provider, or deployment platform and still demonstrate that it interprets the same Contract and Passport rules. The strongest next step for Version 1.0 is an independently written implementation that produces the same decisions without importing the reference runner.

9.5 Local reporting turns the record into operational feedback

The Router's first reporting profile uses the Change Passports already created by the delivery process. It reports authorization blocks, allowed- and protected-scope violations, missing evidence, review-required decisions, repeat evaluations, batch conflicts, risk-tier distribution, and the production history actually recorded in those Passports.

Those measures do not decide that an authorization or policy was good. They expose patterns that deserve governance review: risk-tier distribution, missing-evidence blocks, repeat evaluations, scope violations, and repeated Contract versions become visible over time. The value is not that the framework grades the decision; the record makes the decision and its operational consequences inspectable.

The exact definitions, eligible entries, numerators, denominators, maturity windows, missing-data vocabulary, and report schema belong in the Specification. The whitepaper-level point is simpler: the records are structured enough to support repeatable measurement, and the Router provides one local implementation of those calculations.

Report generation reads a consistent local snapshot, produces an inspectable file and digest, and performs no network access. A team can compare its own periods, inspect where boundaries are repeatedly incomplete, and see whether evidence or batch integrations are creating avoidable friction. Recorded production views remain labeled as recorded history because Passport data alone cannot reveal a deployment or incident for which no Passport exists.

9.6 Optional cohort comparison can add context without exporting the underlying history

A future CEED Benchmark can use the same defined measures to compare similar delivery environments. Participation is separate from local reporting and explicitly optional. Individual Delivery Contracts, Change Passports, source code, repository paths, evidence payloads, deployment identifiers, incident identifiers, and change-level rows remain inside the contributing organization. The contribution is an inspectable aggregate file prepared locally.

Published results should be cohort aggregates, not organization rankings. Context such as production criticality, team size, repository topology, deployment frequency, and AI delivery practice can help readers

avoid comparing unlike environments. Cohorts that are too small or too dominated by one contributor should be withheld under a published methodology.

The comparison remains observational. Participation is self-selected, context cannot remove every confounder, and a lower block rate is not automatically better; it may reflect disciplined work, broad Contracts, weak protection, or some combination. Benchmark results can identify questions worth investigating. They cannot prove that the CEED Framework, a particular tool, or an AI operating mode caused the difference.

Measure locally to improve. Compare by cohort to learn.

The purpose is not to create a league table. It is to help a team see which result deserves investigation and which practices may be worth studying in a comparable environment.

Version 1.0 defines the initial Passport-native reporting and optional contribution shapes. It does not imply that a public Benchmark dataset or outcome report already exists. Broader gate-coverage and production-outcome comparisons require later read-only integrations with repository, release-inventory, and incident-inventory data.

9.7 Version 1.0 is a starting point

Version 1.0 is built for the common, well-defined case the CEED Model is designed to cover. The limits fall into three different kinds, and they should not be treated as the same problem.

The first kind is a roadmap question. Some limits are mechanical, such as merge-queue edge cases or delivery topologies that need additional rules. Others concern fit, such as workflows and team structures that do not map cleanly to the current model. These belong to later versions, informed by implementation experience rather than speculation.

The second kind is readiness. A team that is not yet producing meaningful AI-generated work, does not have reliable Git and CI discipline, or cannot identify accountable owners will gain less from the framework. That is a timing and foundation question, not necessarily a defect in the model.

The third kind is outside CEED's intended scope. The CEED Framework enforces and records authorization decisions a team has learned to make. It does not teach the organization how to choose its architecture, define roles, create good requirements, or build the discipline needed to make those decisions well. Its job is the enforcement and the record, not the organizational maturity underneath them.

What Version 1.0 establishes

The current Version 1.0 design is concrete, testable, and executable for the included cases. Independent interoperability, production readiness across environments, adoption effort, and measured business or safety outcomes remain evidence questions for implementation and field use.

Closing

One authorization history from requirement through production

AI changes the economics of software production by allowing more work to be created than people can inspect line by line, including technically successful changes that fall outside the repository scope anyone intended. Human review therefore cannot be the sole mechanism for verifying change scope. Human intent must be turned into a machine-readable authorization that the delivery system enforces at the configured gates.

The Delivery Contract records that authorization before implementation begins. The Router checks the exact candidate at merge and the approved rule at each required rollout checkpoint. The Change Passport preserves the Contract versions, decisions, evidence references, deployment history, and production knowledge as the record grows.

Git, CI, security, deployment, observability, and incident systems continue to perform their specialist work. The CEED Framework does not compete with them. It gives their outputs one per-change authorization and one portable history, so a later reader does not have to reconstruct the governing decision from the artifacts it produced.

That history first serves the team that owns it. It shows what was approved, whether the candidate stayed inside the approved repository scope, which evidence supported the decision, and how the release advanced.

It also records what was learned afterward. Local reporting turns the same records into a repeatable cycle of inspection and improvement.

Where organizations later choose to contribute aggregates, the same definitions can support cohort comparison without publishing the underlying authorization records. Conformance allows independently built implementations to demonstrate agreement with the defined model.

The underlying mechanisms are established. The CEED Framework's contribution is the standard that binds them: authorization before production, evidence before progression, and a record that remains understandable after the tools and people around it have changed.

References and companion documents

Research and regulation cited

[1] Qu, Y. et al. (2026). Overeager Coding Agents: Measuring Out-of-Scope Actions on Benign Tasks. arXiv:2605.18583.

[2] Yan, Z. et al. (2026). Do Coding Agents Understand Least-Privilege Authorization? arXiv:2605.14859.

[3] Mellafe Zuvic, D. (2026). Capability Gates Are Not Authorization: Confused-Deputy Failures in LLM Agent Frameworks. arXiv:2606.28679.

[4] European Parliament and Council of the European Union. (2024). Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act). Official Journal of the European Union.

Planned Version 1.0 companion documents

The documents below are planned for the Version 1.0 package and are not publicly released with this Public Review Draft.

Document	Role
CEED Version 1.0 - Start Here	Concise release entry point, evidence status, and package navigation
CEED Specification and Change Passport Schema	Normative model rules, structures, path profile, gate behavior, reporting interfaces, validation rules, and conformance requirements
CEED in Practice	Complete REQ-1432 lifecycle with stored artifacts, late incident, corrective cycle, batching cases, and reporting output
CEED Tool Verification Matrix	Dated primary-source evidence, challenge set, and potential implementation roles for adjacent technologies
CEED Conformance Pack	Machine-readable schemas, semantic checks, behavioral vectors, examples, report fixtures, and golden outputs

Stewardship

The CEED Project maintains the working Specification, Change Passport Schema, Tool Verification Matrix, conformance materials, and initial Passport-native reporting definitions as the Version 1.0 package is prepared. Broader Benchmark methodology and reporting profiles may follow when the required integrations, privacy rules, and participation exist. Questions, suggestions, and corrections can be submitted through the CEED website. The CEED Framework remains an independently authored public standardization proposal; the author is the decision authority for project releases, and accepted changes and compatibility implications will be recorded in versioned releases.