
FRAMEWORK · COMPANION · V1.0

Code-as-compliance

And how PDCA+ positions itself

Bearing south · Fourth piece · PDCA+ v2.0 · Public Review

Compliance rules expressed as executable code, evaluated continuously against running systems, governed through the same engineering rigour as application code — that is code-as-compliance. PDCA+ does not invent it, but the framework's structural commitments are what give code-as-compliance the substrate it needs to do its most valuable work.

AUTHOR

Joacim Brandell

Builds on the entire South bearing · honest about what code-as-compliance is and is not · positions PDCA+ as substrate, not replacement

The closing piece

This is the fourth and final piece of the South bearing. Infrastructure foundations named the physical layers. The substrate in detail specified the data model, governance, authority structure, and adoption path. Observing and integrating the substrate established the production face — what needs to be observable, the API surface that emerges from observability needs, the integration patterns that follow, and the hard parts that have to be planned for. With those foundations in place, the question that remains is what runs on top — and that is the question code-as-compliance answers.

Code-as-compliance is a real movement with its own conventions, tooling, and momentum. It has emerged from policy-as-code work (Open Policy Agent, HashiCorp Sentinel, Cedar), from compliance-automation platforms (Drata, Vanta, Anecdotes, Tugboat Logic), from infrastructure-as-code traditions (Terraform, Pulumi), and from the broader recognition by both compliance professionals and engineers that compliance documents that are not executable are compliance documents that drift from the systems they describe. The movement exists independently of PDCA+. This piece does not invent it; it describes it accurately, then locates the framework precisely within it.

The framing this piece takes: code-as-compliance is valuable on its own and growing rapidly with or without PDCA+. What PDCA+ contributes is the structural substrate on which code-as-compliance can do its most valuable work. Without the substrate, code-as-compliance is a collection of rules evaluating against whatever data happens to be available — usually fragmented across operational systems, with classifications and translations done ad-hoc by the rules themselves. With the substrate, code-as-compliance becomes rules evaluating against a coherent representation of the organisation's compliance state, where the classifications and translations are themselves substrate content the rules can rely on. Neither half is the other's prerequisite, but together they produce something neither could produce alone.

This piece walks through what code-as-compliance is, distinguishes it from its precursors, names the criteria that make a compliance rule executable, outlines the CI/CD pipeline that compliance code needs, develops drift detection as the central operational capability, locates PDCA+ precisely within the movement, names the cultural shift that is underway, and is honest about what code-as-compliance does and does not do. A reader who finishes the piece should understand the movement clearly and know where the framework's contribution begins and ends.

What code-as-compliance is, and where it came from

Code-as-compliance is the practice of expressing compliance rules as executable artefacts — code that can be evaluated against running systems, that produces deterministic answers, that lives in version control, that goes through testing and review, and whose evaluation history is auditable. The rules themselves describe what compliant configuration or

behaviour looks like; the executable form lets a machine check whether actual configuration or behaviour matches.

The movement has multiple sources. Policy-as-code, emerging from cloud-infrastructure governance, demonstrated that infrastructure decisions could be governed by declarative rules — Open Policy Agent and Rego becoming the most widely-adopted exemplars. Infrastructure-as-code, from the DevOps tradition, established that infrastructure itself could be specified declaratively in tools like Terraform, with the resulting state continuously reconciled against declared intent. Compliance-automation platforms emerged in the security and audit space, focusing initially on SOC 2 and similar attestation regimes, demonstrating that significant portions of compliance evidence collection and verification could be made continuous and automatic. Each of these traditions contributed pieces of what is now beginning to coalesce into code-as-compliance as a broader practice.

What distinguishes code-as-compliance from its precursors is the explicit recognition that compliance — not just configuration, not just infrastructure, not just narrow security policies — can be brought into the same engineering discipline that other code receives. Compliance rules become artefacts that are written, reviewed, tested, deployed, monitored, and retired through the same lifecycle that application code goes through. The compliance function and the engineering function start to share vocabulary, tooling, and practices.

WHAT CODE-AS-COMPLIANCE IS, IN ONE SENTENCE

Compliance rules expressed as executable code, evaluated continuously against running systems, governed through the same engineering rigour as application code — emerging from policy-as-code, infrastructure-as-code, and compliance-automation traditions, but distinguished by treating compliance itself as code rather than as documentation or configuration.

Three things conflated, distinguished

Code-as-compliance gets conflated with two adjacent things that are not quite the same. Distinguishing them helps the reader see what the movement actually offers.

Compliance documents are the historical form — policies, procedures, standards, control specifications written as prose and stored as files in document management systems. Documents describe what compliance is supposed to look like. They are read by humans, interpreted by humans, and the comparison between what the documents say and what the systems actually do is performed by humans, periodically, in compliance reviews and audits. The drift between document and reality accumulates between reviews.

Compliance configuration is the intermediate form — settings, parameters, control values that live in compliance tooling (GRC platforms, configuration databases, control inventories) as structured data. Configuration is machine-readable but not executable; it describes desired state in a form a machine can parse, but the comparison between desired state and actual

state still happens through reports, dashboards, and human review. Configuration narrows the drift between intent and reality but does not eliminate it.

Compliance code is the contemporary form — rules that execute, evaluate, and produce results without human intermediation. Code can be tested, deployed, versioned, and run continuously. The comparison between intended state (the code) and actual state (the systems being evaluated) happens by execution, not by reading. Code-as-compliance is the movement that treats the third form as the primary form, with documents and configuration as supporting artefacts rather than as the locus of compliance work.

Dimension	Documents	Configuration	Code
Form	Prose, structured	Structured data	Executable rules
Read by	Humans	Humans and machines	Machines (with human review of source)
Comparison method	Manual review	Reports and dashboards	Continuous execution
Drift between intent and reality	Large; accumulates between reviews	Moderate; bounded by reporting cadence	Minimal; bounded by execution frequency
Auditability	Document version control	Configuration change logs	Code review history + execution traces
Testing	Walkthroughs, walkthroughs	Configuration validation	Unit tests, integration tests, regression suites
Cost of change	Document update + retraining	Configuration update + re-deployment	Code change + CI/CD pipeline

The progression is not strictly linear in practice — most organisations operate in all three forms simultaneously, with the mix evolving over time. Compliance documents do not disappear when code-as-compliance is adopted; they remain as the human-readable expressions of compliance intent, the source from which executable rules are derived. Configuration remains for things that genuinely are settings rather than rules. What changes is which form is the primary locus of compliance work, and the answer increasingly is code for the kinds of compliance that can be expressed executably.

THE PROGRESSION, IN ONE SENTENCE

Documents describe compliance; configuration specifies it; code executes it. The three forms coexist, but code-as-compliance shifts the primary locus of compliance work into the executable form, with documents and configuration becoming supporting artefacts rather than the work itself.

What makes a compliance rule executable

Not every compliance requirement can be expressed as code. Some can. Distinguishing them is the first practical move in adopting code-as-compliance. A compliance rule is genuinely executable when it satisfies five criteria.

Deterministic. Given the same inputs, the rule produces the same answer. There is no ambiguity in how the rule applies to a given situation. "All production databases must use encryption at rest with AES-256 or stronger" is deterministic — given a database and its encryption configuration, the answer is yes or no. "The organisation must demonstrate appropriate security culture" is not deterministic — what counts as "appropriate" requires judgement that no rule can encode.

Evaluatable against accessible data. The rule's inputs must be data that can be retrieved by the evaluation engine — either through APIs to operational systems, through the substrate's state, or through the rule engine's own data sources. A rule that depends on data only available through human inspection cannot be executable, regardless of how clearly it is specified.

Declarative, where possible. Compliance rules expressed declaratively ("this property must hold") are more maintainable than rules expressed imperatively ("do these steps to check"). Declarative rules separate what is being checked from how the check is performed, which makes the rules easier to read, review, test, and evolve. Some rules are unavoidably procedural; most are not.

Testable. The rule must be capable of being tested against synthetic inputs that establish both passing and failing cases. A rule that cannot be tested cannot be safely changed, because the consequences of a change are not predictable. Test coverage for compliance rules is no different in principle from test coverage for application code.

Auditable in evaluation. When the rule produces a result, the result must be traceable back to the specific inputs that produced it and the specific version of the rule that was applied. The evaluation history matters for both regulatory audit and for diagnosing rule changes that produce unexpected results.

Compliance requirements that meet these criteria can be expressed as code. Those that do not — requirements that fundamentally require judgement, that operate on data inaccessible to evaluation, or that resist deterministic articulation — remain in document and configuration form. Code-as-compliance does not claim every compliance requirement should be expressed as code; it claims the ones that can be, should be, because the operational benefits of execution over reading are substantial.

A simple example

The following is an illustrative policy-as-code rule, written in a syntax similar to Rego (OPA's policy language), for a compliance check against an organisation's database configurations.

```
package compliance.encryption_at_rest

# Rule: All production databases must use AES-256 or stronger
# Maps to: Control Objective CO-ENC-001
# Maps to: CMMC MP.L2-3.8.1, ISO 27001 A.10.1.1

default allow := false

allow {
    input.database.environment == "production"
    input.database.encryption.enabled == true
    input.database.encryption.algorithm in ["AES-256", "AES-384", "AES-512"]
}

deny[reason] {
    input.database.environment == "production"
    not input.database.encryption.enabled
    reason := sprintf("Production database %v has encryption disabled",
[input.database.name])
}

deny[reason] {
    input.database.environment == "production"
    input.database.encryption.enabled
    not input.database.encryption.algorithm in ["AES-256", "AES-384", "AES-512"]
    reason := sprintf("Production database %v uses weak algorithm: %v",
[input.database.name, input.database.encryption.algorithm])
}
```

An illustrative Rego-style rule. The mappings in comments link the rule to the Control Objective and to the regime references — substrate addressing made explicit in code.

The rule is deterministic (same inputs produce the same answer), evaluable against accessible configuration data, declarative in form, testable through synthetic inputs, and auditable both in source (through code review) and in evaluation (through trace logging). It maps explicitly to a Control Objective Record and to the regime requirements that objective satisfies — which is where PDCA+'s substrate begins to do its work, but more on that in §5.

THE FIVE CRITERIA, IN ONE SENTENCE

Deterministic, evaluable against accessible data, declarative where possible, testable, and auditable in evaluation. Compliance requirements meeting these criteria can be expressed as code; those that cannot remain in document or configuration form. Code-as-compliance does not claim everything should be code.

The CI/CD pipeline for compliance code

Code-as-compliance requires the same pipeline discipline as application code. The stages are familiar to anyone with modern engineering experience; what differs are the compliance-specific concerns at each stage. Naming them explicitly is what allows the pipeline to be designed deliberately rather than improvised.

Stage	What happens	Compliance-specific concerns
Author	Compliance rule is written in source, with mappings to Control Objectives and regime references documented in code	The rule's author must understand both the regime requirement and the operational reality being evaluated; rule design is collaborative across compliance and engineering
Test	Rule is tested against synthetic inputs covering passing cases, failing cases, edge cases, and regression cases	Tests must reflect the regime's actual expectations; tests are themselves substrate content that must be governed
Review	Rule changes go through code review by qualified reviewers before merge	Review requires both engineering and compliance perspectives; the authority to approve compliance rule changes must be granted to specific identities or roles
Stage	Rule is deployed to staging environment and evaluated against staging substrate state	Staging must include representative substrate state — not just operational state — so the rule's interaction with substrate content is observable before production
Deploy	Rule is deployed to production with monitoring for unexpected results	Deployment must record a Substrate Change Record with the rule version, authority invoked, and rationale; rollback must be available
Monitor	Rule's evaluations are observed continuously for drift, anomalies, and unexpected failure patterns	Evaluations contribute to second-order observability — the framework can detect when its own rules are no longer adequate to the reality being evaluated
Retire	Rule is marked for retirement when the underlying requirement is superseded or no longer applicable	Retirement requires the same governance as deployment; retired rules' evaluation history must remain auditable

What the pipeline gives compliance

Most of what the pipeline gives is what software engineering has had for years: predictability of change, the ability to test before deployment, traceability of decisions, the discipline of code review. For compliance, these capabilities have historically been weak. Compliance documents are revised through document-management workflows that are review-heavy but test-light. Configuration changes propagate through GRC platforms whose change-management is rudimentary compared to engineering CI/CD. The pipeline brings compliance into a discipline that has been refined for application code over decades.

What is specific to compliance is the governance layer overlaid on the pipeline. Authority to author, review, deploy, and retire compliance rules has to be explicit; the Authority Record entities from the substrate-in-detail piece are what governs who can do what at each stage. The Substrate Change Record entities capture each pipeline event as substrate content. The result is a pipeline that is operationally efficient like engineering CI/CD and structurally

auditable like compliance governance demands. Neither tradition gets compromised; both contribute their strengths.

THE PIPELINE, IN ONE SENTENCE

The seven stages familiar from engineering CI/CD — author, test, review, stage, deploy, monitor, retire — applied to compliance code with substrate-specific governance overlaid. The result is operationally efficient like engineering pipelines and structurally auditable like compliance governance demands; neither tradition gets compromised, both contribute their strengths.

Drift detection as the central capability

Code-as-compliance's most valuable operational property is drift detection — the continuous comparison between intended state (what the rules say) and actual state (what the systems being evaluated actually look like). Drift detection is what turns compliance from periodic verification into continuous assurance, and it is the capability that justifies the engineering investment code-as-compliance requires.

Drift, in this context, is the divergence between what compliance rules expect to find and what they actually find when they execute. A drift may indicate that an operational system has changed in a way the compliance rule needs to be updated for; or that the rule itself is wrong about what it should be checking; or that the system is genuinely non-compliant and someone needs to act on the finding. The continuous-execution discipline of code-as-compliance is what makes the drift visible quickly enough to be actionable.

Three kinds of drift

Drift detection identifies three distinct categories, each requiring different response.

Operational drift. The operational system has changed in a way that violates the compliance rule. A database has been deployed without encryption; a network rule has been added that exposes a sensitive port; a service account has been granted excessive permissions. The drift is in the system; the rule is correct; the response is to fix the system. This is what most discussions of compliance monitoring focus on, and it is the most operationally common form of drift.

Rule drift. The compliance rule has become inadequate to the regime requirement it is supposed to enforce. The regime has revised; the operational landscape has shifted; the rule's assumptions are no longer accurate. The drift is in the rule; the system may or may not be compliant; the response is to update the rule. This category is harder to detect because the rule cannot detect its own inadequacy — but second-order observability, computed across many rule evaluations and their outcomes over time, can surface patterns suggesting rule drift.

Reality drift. The compliance rule and the operational system both reflect a reality that no longer applies. The control objective the rule supports has been superseded; the regime has

been retired; the underlying risk landscape has shifted such that the original concern is no longer the right one to address. Both rule and system may be "correct" in the narrow sense, but the work they represent has lost its grounding. This is the rarest and most consequential category, and it is detectable only by second-order observation of why specific rules exist and what they are supposed to be defending against.

Operational drift is what conventional compliance monitoring detects. Rule drift and reality drift require the substrate, the chain architecture, and the second-order observability the previous South bearing piece established. This is one of the places where PDCA+'s structural commitments produce specific operational value that code-as-compliance alone does not.

What drift detection enables

Drift detection done continuously enables a different relationship between compliance and operations. Compliance no longer arrives as a periodic verification event; it is a continuous property of the operational state. When systems pass the rules, compliance is current. When systems fail the rules, the failure is surfaced within minutes rather than discovered at the next audit. The compliance function shifts from after-the-fact verification to before-the-fact assurance, with the time between drift and detection collapsed.

This is the continuous-compliance mode the West bearing (Regimes) pieces described. Drift detection is the technical mechanism that makes continuous compliance operationally real. Without it, continuous compliance is an aspiration. With it, continuous compliance is a property of the operating system that produces compliance evidence as a byproduct of normal operation, with audit windows becoming snapshots of an ongoing reality rather than reconstructions of historical state.

DRIFT DETECTION, IN ONE SENTENCE

Continuous comparison between intended state (rules) and actual state (systems) produces three categories of drift — operational (system changed), rule (rule inadequate), and reality (both have lost their grounding) — with operational drift detected by code-as-compliance directly and the other two requiring the substrate and second-order observability the framework provides.

Where PDCA+ positions itself

With code-as-compliance described as its own movement, the question of what PDCA+ adds becomes precisely answerable. Five contributions are worth naming, each grounded in structural commitments the earlier pieces established.

The substrate as evaluation target

Code-as-compliance rules need something to evaluate against. Without the substrate, the rules typically evaluate against raw operational data — system configurations, cloud account state, application telemetry — which fragments across operational systems and requires each

rule to know how to retrieve and interpret its own evidence. With the substrate, rules evaluate against a coherent representation that already holds classifications, translations, regime mappings, and current state. The rules become simpler because the substrate has already done the harmonisation work; the rules become more powerful because they can reason about compliance-meaningful concepts rather than only about raw configuration.

The chain architecture as rule addressing

Rules in code-as-compliance need to be addressable to the compliance work they support. Without the chain, rules typically address themselves to control implementations, which fragments the rules along with the implementations across regimes. With the chain, rules can be addressed to Control Objective Records — the regime-portable layer — which makes the same rule serve multiple regimes when their objectives converge. The rule example in §3 showed this with the comment-line mappings; in production the mappings are first-class metadata that the substrate uses to determine which rules are relevant to which regime requirements.

The conjunction as continuous-evaluation home

Code-as-compliance needs continuous evaluation to deliver its continuous-assurance promise. The conjunction's continuous observation of the substrate is the natural home for this evaluation. Rules run as substrate observers; their results become substrate findings; findings flow through the same workflow and routing infrastructure as all other substrate content. Code-as-compliance becomes part of the substrate's ongoing operation rather than a separate evaluation pipeline that has to integrate with the substrate externally.

Second-order observability as rule governance

Code-as-compliance produces a body of rules that itself needs to be observed. Are the rules still adequate? Are they being applied correctly? Are they producing patterns that suggest rule drift or reality drift? Conventional code-as-compliance has no structural answer to these questions; the rules are governed through code review and changelog discipline but not through observation of their actual behaviour over time. The framework's second-order observability is what closes this gap. The substrate observes how its own rules are operating, surfaces patterns that suggest inadequacy, and treats the rule body as something that itself can drift relative to the regimes it is supposed to enforce.

The authority structure as deployment governance

Code-as-compliance pipelines need governance for what gets deployed, by whom, under what authority. Conventional engineering CI/CD has code review and access control but does not natively distinguish between roles, authorities, and accountabilities the way compliance governance requires. The substrate's identity-authority-accountability structure maps onto the CI/CD pipeline cleanly — authorities are granted to deploy specific rule categories; the Substrate Change Record captures each pipeline event with the authority invoked; the conjunction observes whether deployments are made under appropriate authority. The

result is a deployment pipeline that satisfies both engineering rigor and compliance governance simultaneously.

Taken together, these five contributions are what PDCA+ adds to code-as-compliance. None of them is something the movement could not eventually develop on its own; the framework simply provides them as integrated structural commitments rather than as point capabilities to be assembled. The framework does not invent code-as-compliance; it makes code-as-compliance structurally sound.

WHAT PDCA+ ADDS, IN ONE SENTENCE

Substrate as evaluation target, chain architecture as rule addressing, conjunction as continuous-evaluation home, second-order observability as rule governance, and the authority structure as deployment governance — five contributions integrated structurally, none of which code-as-compliance has natively but each of which it benefits from substantially.

The cultural shift, named

Code-as-compliance is not only a technical change; it is a cultural one. The shift is worth naming directly because it affects the people doing the work, and naming it explicitly helps both sides recognise what is happening to them.

Compliance professionals who have spent careers as document authors and assessment runners are becoming substrate stewards and rule authors. The skills required are not entirely different — the deep regime knowledge they have remains essential — but the form in which they apply it is shifting. They are no longer writing policy documents that someone else may or may not implement; they are writing rules that execute. They are no longer reviewing assessments that someone else has prepared; they are observing rule evaluations and second-order drift in real time. The work is more leveraged, more visible, and more closely coupled to operational reality. It is also less ceremonial — the rituals of compliance documents lose some of their centrality.

Engineers who have spent careers ignoring compliance — treating it as an interruption to engineering work or as overhead imposed by people who do not understand systems — are encountering compliance differently. Compliance arrives as code that has to be written, tested, and deployed alongside their other code. They cannot dismiss it as someone else's concern because the code is in their repositories, their pipelines, their dashboards. They have to engage with the regime intent the rules express, not just with the rules themselves, because intent is what they need to understand to write good rules. The encounter is sometimes uncomfortable, but it produces engineers who understand compliance and compliance professionals who understand engineering. Both sides learn things they did not previously know.

The cultural shift is asymmetric in one important way. Compliance professionals are the ones who have to learn a new toolchain, often a new vocabulary, sometimes new abstractions. Engineers are largely adding compliance to a toolchain they already know — code review,

CI/CD, testing, observability — and the cognitive load is lighter for them. Recognising this asymmetry helps organisations support the compliance professionals making the larger transition, rather than expecting both sides to shift equivalently.

There is also a new kind of role emerging — the practitioner who is comfortable in both worlds, who can write a compliance rule that an engineer will accept and a regime mapping that a compliance lead will recognise. These hybrid practitioners are rare and increasingly valuable. Many organisations call them "GRC engineers" or "compliance engineers" or "security engineers with compliance scope." The titles are not standardised yet; the role itself is forming in real time across the industry. New-era developers entering this space are, in many cases, this role — they treat compliance as a domain that benefits from the same engineering practices they apply to everything else, and the framework's structural choices are what allow them to do their best work.

THE CULTURAL SHIFT, IN ONE SENTENCE

Compliance professionals become substrate stewards and rule authors; engineers begin engaging with compliance because it appears as code in their repositories; a new hybrid role — variously called GRC engineer, compliance engineer, or new-era developer — is emerging at the intersection. The shift is real, asymmetric, and productive.

Limits and honest framing

Three things to acknowledge before closing.

First, code-as-compliance does not eliminate the need for judgement. The criteria in §3 named what makes a rule executable — and named, by implication, what cannot be made executable. Cultural compliance, ethical compliance, contextual judgement about what "appropriate" or "reasonable" means in specific situations — these resist code expression for structural reasons. The piece does not argue these should be expressed as code; it argues that the substantial fraction of compliance work that can be expressed as code, should be, because the operational benefits are real. The remainder — judgement, culture, ethics, context — continues to need humans, and the framework's structural commitments support that work alongside the executable work rather than displacing it.

Second, code-as-compliance does not eliminate compliance professionals. The shift named in §7 changes what they do but does not reduce the need for them. The substrate has to be stewarded; the rules have to be authored by people who understand regimes; the second-order observations have to be interpreted by people who can distinguish drift that requires response from drift that is benign. Organisations adopting code-as-compliance who expect to reduce their compliance workforce typically discover the workforce is needed in different ways, with different skills, in roughly similar total numbers. The cost structure of compliance changes; the cost itself does not necessarily decrease, at least not in the short term.

Third, code-as-compliance is not universally appropriate for all regimes. Some regimes specify so much in terms of human judgement and process that the executable portion is

small. Some regimes are revised so frequently that the rule-maintenance cost exceeds the benefits. Some regulatory environments are not yet ready to accept evidence in the forms code-as-compliance produces. Adoption should be guided by where the benefits are highest — typically in regimes with technical content, frequent operational drift, and substantial overlap with other regimes — and tempered by where the benefits are lower. The framework supports both, but the practitioner choosing where to invest should be honest about where the returns are.

None of these limits diminishes what code-as-compliance offers in the contexts where it does offer value. They establish, instead, the conditions under which the offering is genuine. A practitioner who knows these limits will deploy code-as-compliance well, and the deployment will produce the operational benefits the movement promises. A practitioner who ignores the limits will produce a code-as-compliance implementation that overreaches, fails predictably, and discredits the broader practice. Honest framing is what allows the genuine value to be realised.

Closing the South bearing

The South bearing of this series has been four pieces. Infrastructure foundations named what physical infrastructure PDCA+ requires — the seven layers from substrate core to artefact storage, with the GRC tool positioned as operational-artefact-of-record alongside the framework's other components. The substrate in detail specified the data model, the lifecycle and governance of substrate content, the identity-authority-accountability structure that grounds the governance, and the five-stage adoption path that takes organisations from where they are today to strategic compliance. Observing and integrating the substrate established the production face — four observability requirements, three pillars applied with substrate-specific meaning, second-order observability as the framework's own work, six API categories that emerge from observability needs, three integration patterns, and seven hard parts named honestly. This fourth piece has developed code-as-compliance as the methodology that runs on top of the entire stack — what it is, what makes a rule executable, the CI/CD pipeline it requires, drift detection as the central capability, the five contributions PDCA+ adds, the cultural shift underway, and the limits that honest framing has to acknowledge.

Taken together, the four pieces compose into a buildable operationalisation specification — not the specification, because tool choices and organisational adaptations remain with the implementing organisation, but a coherent foundation that an architect, GRC technology lead, enterprise architect, technically-oriented CISO, or new-era developer can work from. The journey on the South bearing is now complete in the sense that someone wanting to operationalise PDCA+ has the technical roadmap they need; what remains is implementation, which is by design left with the organisations doing the work.

WHAT THIS PIECE ESTABLISHED, IN ONE SENTENCE

Code-as-compliance as a real movement with its own roots and momentum; the five criteria that make a rule executable; the seven-stage CI/CD pipeline with substrate-specific governance; drift detection as continuous assurance with three categories of drift; five specific contributions PDCA+ adds without displacing the movement; the cultural shift of compliance professionals and engineers converging on a hybrid role; and the limits that honest framing requires. The fourth piece of the South bearing, and the close of the direction.