FRAMEWORK · COMPANION · V1.0

Operationalising PDCA+

Infrastructure foundations

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

What it takes to operationalise PDCA+ in real infrastructure — the database the substrate runs on, the workflow engine that routes findings, the event flow that makes continuous observation possible, the GRC tool's actual role, and how the pieces fit together as a reference architecture.

AUTHOR

Joacim Brandell

Written for technical architects, GRC technology leads, enterprise architects, and technically-oriented CISOs · tool-agnostic in principle, specific enough to be implementable

Why this piece, and why first

This piece is the foundation of the south bearing. It walks through what PDCA+ requires of real infrastructure — the database, the workflow engine, the event-flow architecture, the GRC tool's actual role, the data catalog and policy-as-code layer, the document store — and ends with a reference architecture that ties the pieces together. It is tool-agnostic in principle: PDCA+ does not require any particular vendor or technology. It is specific enough in practice that a technical architect could begin scoping an implementation by the end of it.

One distinction matters from the start, and naming it now prevents confusion later. The PDCA+ framework talks extensively about "schemas" — schemas as the conjunction's structural definitions of what flows between disciplines. Databases also have schemas, in the DDL sense — tables, columns, relationships, constraints. Both are legitimately called schemas. The conjunction's schemas are partially realised in database schemas, but the conjunction's substrate is bigger than the database. It includes things that live in workflow engines, integration buses, policy-as-code stores, document repositories, even runbooks and conventions. Throughout this piece, "conjunction schema" refers to the structural commitment; "database schema" refers to the technical realisation. Where the two need to be distinguished, both terms are used.

The database, and what actually goes in it

The substrate's core requires persistent, queryable, relational (or graph-relational) storage. This is the database layer. It is the foundation everything else builds on, and what goes in it is more than most implementations initially scope for.

What the substrate database needs to hold

The chain records from the regime architecture. Regime Intent, Requirement, Control Objective, Obligation, Evidence Link, Stakeholder Entitlement — the six record types the control-objective chain appendix described. Each is a first-class persistent entity with its own identity, lifecycle, relationships to other records, and audit history. The Control Objective Record is particularly load-bearing because it is the chain's fulcrum; queries against it have to be efficient and its relationships have to be traversable both upward (toward regime requirements) and downward (toward obligations).

Discipline bindings. For each bound discipline, a record naming which strands it contributes to, which conjunction services it consumes, what dimensional axis it operates on, and how its inputs and outputs flow. These are reference data the routing layer queries continuously; they have to be retrievable in microseconds, not milliseconds.

The control inventory. One inventory, held once, addressed by every regime that references it. Each control has its specification, its current implementation description, its measurement constructs, its effectiveness observations, and its mappings to control-objective records. This is the framework's most consequential shared capability; if there is one piece of substrate that must not fragment, it is this.

Operational artefact records. Risk register entries, configuration items, incident records, change records, audit findings — the artefacts each discipline produces and consumes. These are the operational records, and they typically already exist in some form in most organisations; the substrate's contribution is holding them in shapes that the conjunction's schemas can address and the chain can connect to.

Taxonomies and classifications. Reference data that other records are classified under. Configuration-deviation categories, incident severity scales, risk taxonomies, regime classification schemes. Held as queryable reference data with versioning, because taxonomies revise and the substrate has to remember what classification was current when each historical record was classified.

Translation rules. Mappings between standards' vocabularies — "risk owner" under ISO 31000 to "Authorizing Official" under NIST RMF, control reference mappings between CMMC and DEFSTAN, evidence-form mappings across regimes. Each rule has scope conditions, caveats, and the operational rules of when it applies. Translation rules are queried by every regime addressing the substrate from its own side.

Measurement constructs. The shared definitions of adequacy, severity, effectiveness against which strands observe. These have to be conjunction-held rather than discipline-held,

because the whole point is that they are shared. ISO 27004 and NIST SP 800-55 articulate the form; the database has to hold the specific constructs the organisation operates against.

Audit trail of substrate changes. Every change to schemas, taxonomies, translation rules, control specifications, regime bindings — recorded with timestamp, actor, and rationale. This is what makes the conjunction's second-order observation possible: detecting drift requires knowing what changed when. Without an audit trail of substrate changes, the framework cannot observe its own observing apparatus.

Relational, graph, or both

The records and their relationships argue for both relational and graph patterns. Relational works well for the operational artefact records — they have stable shapes, predictable queries, transactional updates. Graph works better for the chain relationships — Regime Intent → Requirement → Control Objective → Obligation → Evidence is structurally a graph traversal, and queries like "what evidence underwrites this regime's intent?" are graph queries that relational schema handles awkwardly.

Most real implementations end up with both. A relational database (Postgres, SQL Server, MySQL) holds the operational artefacts; a graph database (Neo4j, Amazon Neptune) holds the chain and the cross-cutting relationships; the two are kept consistent through event-driven synchronisation. Some organisations use a single multi-model database that supports both patterns (ArangoDB, OrientDB) and accept the operational trade-offs. The choice matters less than getting the data model right; the data model is what determines whether queries against the substrate are tractable.

> THE DATABASE, IN ONE SENTENCE
>
> *The substrate's core requires persistent storage for chain records, discipline bindings, control inventory, operational artefacts, taxonomies, translation rules, measurement constructs, and the audit trail of substrate changes themselves — typically as a combination of relational storage for operational data and graph storage for chain relationships.*

The workflow engine: where routing lives

The whitepaper described findings flowing between disciplines without human intervention — event management's output reaching configuration management through the substrate, not through a runbook a person reads and acts on. That structural property has a specific technical home: the workflow engine.

Workflow engines (n8n, Camunda, Temporal, Argo Workflows, AWS Step Functions, Azure Logic Apps) hold the executable logic that turns substrate-state changes into routed actions. When an operational discipline produces an output that is classified under a taxonomy and conforms to a schema, the workflow engine queries the binding declarations of other bound disciplines, identifies which of them should receive the output as input, and routes it. The

routing happens because the workflow engine reads the substrate's binding declarations and acts on them; nothing is implicit.

What the workflow engine specifically contributes

- Stateful long-running processes. Many cross-discipline interactions are not request-response; they unfold over hours, days, or weeks. A finding that needs review by risk management, then treatment by configuration management, then acceptance by a named authority, then verification by audit — that is a workflow with state, branches, timeouts, retries, and human-in-the-loop steps. Workflow engines are built for this; databases alone are not.
- Declarative routing based on substrate state. The routing logic is not hardcoded; it is computed from binding declarations and taxonomy classifications held in the substrate. When a binding changes, the routing changes; when a taxonomy revises, routing updates accordingly. The workflow engine becomes a substrate-driven router.
- Human-in-the-loop integration. Some strands require human judgement at specific points — evaluation, acceptance, certain treatment decisions. The workflow engine integrates the human steps into the substrate's automated flow without losing the audit trail of who decided what when.
- Compensation and error handling. When a step fails or a decision needs to be reversed, the workflow engine handles the compensating logic — backing out partial state, notifying affected parties, rolling forward where possible. The substrate's continuous operation depends on these failures being handled gracefully rather than leaving inconsistent state.

A common implementation pattern: n8n or a similar low-code workflow tool for the simpler routing and notification flows (where the value is the speed of authoring), Camunda or Temporal for the complex long-running orchestrations (where the value is the durability and the modelling discipline). Some organisations build their own on top of a state machine library when their requirements are unusual; this works but is more expensive than most teams realise.

The event flow: making continuous observation real

"Continuous observation" is one of the phrases that recurs throughout the PDCA+ literature, and it sounds simple until you try to implement it. The framework requires the conjunction to observe what flows through it, to surface inadequacies as they arise, to make findings legible across disciplines in real time. None of this is possible with batch-oriented data movement. It requires events flowing through observable channels.

The event bus (Apache Kafka, AWS Kinesis, NATS, RabbitMQ, Azure Event Hubs) is where this lives. Operational disciplines emit events as their work proceeds — a new risk identified, a configuration item changed, an incident classified, a change approved. The events flow onto the bus; subscribers consume them; the substrate updates accordingly; the conjunction's observation processes watch the flow itself for patterns that warrant surfacing.

What event-driven architecture enables structurally

Without an event bus, integration between disciplines tends to be either point-to-point (each pair of systems connecting directly to each other, producing N^2 complexity) or batch-oriented (data extracted on schedule, loaded into a warehouse, queried periodically). Both are operationally adequate for many things; neither supports continuous observation in any meaningful sense.

With an event bus, disciplines emit events without knowing or caring who consumes them. The substrate subscribes to the events it needs; the conjunction's observation processes subscribe to all of them. New disciplines can be added without modifying existing producers; new observation patterns can be implemented without modifying existing disciplines. The architecture supports the framework's growth without architectural rework at every step.

Event sourcing — where the database is reconstructed from the event log rather than maintained as a snapshot — is a stronger version of this pattern that some organisations adopt for the audit trail benefits. Every change is an event; the substrate's state at any moment is the result of replaying all events to that moment. This is overkill for most implementations but worth knowing about, because it solves the audit-trail-of-substrate-changes requirement structurally rather than as a separate concern.

> **EVENT FLOW, IN ONE SENTENCE**
>
> *Continuous observation requires events, not batches. The event bus is what lets disciplines emit and the conjunction watch without the two having to know about each other — and it is what lets new disciplines bind without architectural rework.*

Policy-as-code: substrate rules made executable

Some of what the conjunction holds is rules — classification rules that decide what category a finding belongs to, translation rules that say which artefact under one regime corresponds to which under another, coherence rules that detect when two disciplines' outputs are inconsistent, measurement rules that determine when a control's effectiveness is below threshold. These rules live in the substrate as data, but for them to do work, they have to be evaluated by something.

Policy-as-code is the pattern for this. Tools like Open Policy Agent (OPA), HashiCorp Sentinel, Cedar, or Google CEL let rules be expressed in a declarative language, stored as code, version-controlled, tested, and evaluated at runtime against incoming data. When event management emits a finding, a policy-as-code evaluation classifies it under the taxonomy; when configuration management proposes a change, a policy-as-code evaluation checks coherence with current bindings; when the conjunction observes the substrate, policy-as-code evaluations detect classification drift.

What policy-as-code contributes beyond the database

- Declarative rule expression. Rules are stated as conditions in a declarative language rather than as imperative code embedded in applications. The same rule can be evaluated in multiple places (at event ingestion, at substrate query, at audit time) without each place re-implementing it.
- Versioning and audit. Rules are code; they live in a version-control system; their changes are tracked; their evaluations are auditable. When a classification produced a particular outcome, the version of the rule that produced it is recoverable.
- Testing. Rules can be tested against synthetic inputs, regression tests can be run before rule changes are deployed, and the consequences of a rule change can be evaluated against historical data before promotion.
- Separability of policy and engine. The same engine evaluates many rules; new rules are added without code changes to the engine; old rules are retired by removing them from the policy repository. This is exactly the property the conjunction's substrate needs from its rule layer.

A typical implementation uses OPA as the evaluation engine, Rego (OPA's policy language) for the rules themselves, and a policy repository in Git for version control. Rules are deployed to OPA from the repository; evaluations happen at runtime against incoming data; results feed back into the substrate. The pattern is mature enough that organisations adopting it find well-developed tooling, documentation, and community support.

GRC tools: what they actually do, and what they don't

Most organisations adopting PDCA+ already have a GRC tool — Archer, ServiceNow GRC, OneTrust, Hyperproof, Drata, Vanta, MetricStream, ZenGRC, AuditBoard. The question of how the framework relates to these tools matters operationally, because the tools represent significant existing investment and the answer affects both the implementation path and the organisational politics of the change.

The honest answer is that most current GRC tools are operational-artefact holders with attached workflow capabilities. They hold the risk register, the control inventory, the policy documents, the audit findings, the compliance evidence repository, the assessment programmes. They provide workflow for tasks like control assessments, audit-finding remediation, policy attestations, and exception management. Some of them have started to add features that resemble parts of the PDCA+ substrate — control mapping across frameworks, continuous control monitoring integrations, control-effectiveness scoring — but these features are typically additions to an architecture that was not built around the substrate's structural commitments.

Where GRC tools fit alongside PDCA+ infrastructure

GRC tools do well at: **holding operational artefacts.** They were designed for this. Risk registers, control inventories, audit findings, policy documents — these belong in tools that have mature interfaces for compliance professionals, role-based access control, reporting, and integration with regulatory bodies' submission requirements. PDCA+ does not displace this; it depends on it. The operational artefacts the substrate references typically live in the GRC tool.

GRC tools do partially: **control mapping across regimes.** Modern GRC tools include cross-framework mapping features that show how a control in one framework relates to controls in others. These mappings are useful but are not the same as PDCA+'s control-objective-based reuse — they typically map at the control-implementation layer rather than the control-objective layer, which means the mappings fragment as control implementations specialise per regime. The substrate's chain architecture, with its objective-layer fulcrum, replaces these mappings rather than augmenting them.

GRC tools typically do not: **hold the conjunction's substrate.** The schemas, taxonomies, translation rules, and measurement constructs that the conjunction holds are not features of most GRC tools. Some tools hold parts (a taxonomy here, a measurement framework there) but not as a coherent substrate addressable by other components. Building the conjunction's substrate inside the GRC tool is possible but usually fights the tool's architecture.

GRC tools structurally do not: **observe their own observing.** Second-order observation requires the framework to detect drift in its own classifications, schemas, and rules. GRC tools are designed to support first-order operations; their architecture does not include the kind of meta-observation capability the conjunction provides. Adding this to a GRC tool would be a substantial re-architecture rather than a configuration change.

The integration pattern

The pragmatic answer for most organisations is to keep the GRC tool as the operational-artefact-of-record system while building the substrate alongside it. The GRC tool emits events when artefacts are created, modified, or classified; the substrate subscribes to those events through the event bus; the chain records, taxonomies, and translation rules live in the substrate database; the workflow engine routes between the two as appropriate. The GRC tool continues to serve compliance professionals who need its mature operational interfaces; the substrate provides the structural capabilities the GRC tool was not designed to provide.

This pattern recognises that GRC tools are not the framework's replacement target. They are necessary infrastructure that does important work. PDCA+ is the architectural layer that makes the GRC tool's operational artefacts addressable in ways that scale across regimes, surface inadequacies continuously, and support the strategic compliance moves the regime-leg appendices described. Both layers are needed; the substrate is what most organisations are currently missing.

> **THE GRC TOOL'S ACTUAL ROLE**
> *GRC tools are operational-artefact-of-record systems with attached workflow. They hold the risk register, the control inventory, the audit findings. They do not hold the conjunction's substrate, and they do not observe their own observing. PDCA+ infrastructure sits alongside the GRC tool, not in place of it.*

Data catalog and metadata management

The conjunction's substrate has a metadata problem most implementations underestimate. Schemas, taxonomies, translation rules, measurement constructs — these are not data in the operational sense; they are metadata that describes what the operational data means and how it relates. This metadata has to be queryable by humans (so practitioners can understand what classifications exist, what rules apply, what relationships hold) and by machines (so policy engines, workflow engines, and the conjunction's observation processes can resolve references).

Data catalog tools — Collibra, Atlan, DataHub, Apache Atlas, Alation — are designed for this. They hold metadata as first-class entities, support querying and search, provide lineage visualisation, integrate with the systems that produce the underlying data, and present human-readable interfaces alongside machine-readable APIs. For PDCA+ substrate, they are the natural home for the conjunction's non-record metadata: the taxonomies as catalogued reference data, the schemas as catalogued data definitions, the translation rules as catalogued mappings, the measurement constructs as catalogued metric definitions.

What the catalog contributes that the database does not

A relational or graph database can hold metadata as data. A catalog adds discoverability, governance, and lineage. The taxonomy in the database is queryable; the taxonomy in the catalog is also searchable, annotated, browsable through a UI, traceable to its sources, governed by stewards, and linked to the artefacts it classifies. For substrate metadata that has to be understood and maintained by humans across many disciplines, the catalog's affordances matter.

A reasonable pattern is to hold the canonical metadata in the catalog and replicate it (or sync references to it) into the substrate database where operational queries need it. The catalog becomes the source of truth for what the substrate's metadata is; the database becomes the source of truth for what the substrate's records are; the two are kept synchronised through the same event-driven plumbing the rest of the architecture uses.

Document and artefact storage

Operational artefacts that constitute evidence — risk-acceptance memos, configuration baselines, incident reports, policy documents, audit reports, control assessment results — have to be stored somewhere durable, immutable (or at least version-tracked), and retrievable. The substrate's Evidence Link Records point at these artefacts; the artefacts themselves need a storage layer.

Object storage (Amazon S3, Azure Blob Storage, Google Cloud Storage, MinIO for on-premises) is the typical home for artefacts that are essentially files. Document management systems (M-Files, SharePoint, OpenText) suit artefacts that need richer metadata, workflow, or collaboration features. For high-volume operational evidence (logs, configuration snapshots, scan results), specialised systems may be appropriate — log management platforms, configuration-management databases that hold their own historical state, vulnerability-scan repositories.

What the substrate requires of the storage layer is that artefacts be addressable by stable identifier, retrievable by the workflow engine and the policy engine when evidence needs to be presented, and immutable enough that the audit trail of what was used as evidence at what time is reliable. Most modern storage systems meet these requirements; the architectural concern is more about consistent addressing patterns than about choosing the right product.

The reference architecture, assembled

Putting the pieces together: a reference architecture for operationalising PDCA+ has the following layered structure. The layers are functional rather than physical — in many implementations multiple layers share infrastructure, and in others a single layer is split across several tools. What matters is that each functional role is filled.

Substrate core	Persistent storage for chain records, discipline bindings, control inventory, operational artefacts, audit trail of substrate changes	`Postgres + Neo4j or multi-model DB`
Metadata catalog	Schemas, taxonomies, translation rules, measurement constructs — discoverable, governed, lineage-tracked	`Collibra · Atlan DataHub · Atlas`
Policy engine	Declarative rules for classification, translation, coherence checks, measurement evaluation — versioned and tested	`OPA · Sentinel Cedar · CEL`
Workflow engine	Stateful long-running routing based on binding declarations; human-in-the-loop integration	`n8n · Camunda Temporal · Step Fn`
Event bus	Event flow between disciplines; subscription target for conjunction observation processes	`Kafka · Kinesis NATS · EventHubs`
GRC tool	Operational artefacts of record — risk register, control inventory, audit findings, policies	`Archer · ServiceNow OneTrust · Drata`
Artefact storage	Evidence artefacts, documents, snapshots — addressable, durable, audit-immutable	`S3 · Azure Blob M-Files · SharePoint`

How the layers communicate

Events flow between layers through the event bus. The substrate core holds canonical state; the metadata catalog holds canonical metadata; the policy engine evaluates rules against substrate state at points where workflow requires evaluation; the workflow engine orchestrates the longer-running multi-step processes that involve multiple disciplines and human decisions; the GRC tool emits events when operational artefacts change, and reflects substrate state where compliance professionals expect to see it; artefact storage holds the durable evidence the substrate's evidence-link records point at.

The choice of specific products is less important than getting the layer model right. An organisation that has Postgres and Kafka and adopts OPA and n8n can implement PDCA+; an organisation that has Oracle and IBM MQ and adopts Sentinel and Camunda can implement PDCA+. The substrate's structural commitments are realisable on many technology combinations. What matters is that each functional role has a home, that the layers communicate event-driven rather than batch-oriented, and that the substrate is held coherently rather than fragmented across disciplines.

What this isn't

Three honest framings before closing.

First, this piece is not a procurement guide. The specific products named are illustrative examples of what fits each functional role; substituting other products from the same category is generally fine. PDCA+ is tool-agnostic, and any implementation will reflect the organisation's existing technology investments, vendor relationships, and operational preferences. The architecture matters more than the products.

Second, the piece is not arguing that every organisation needs every layer at full sophistication from day one. Real implementations grow incrementally, starting with the smallest substrate that supports the most important bindings and expanding as more disciplines join, more regimes apply, and more sophisticated capabilities become valuable. An organisation can start with Postgres holding the chain records, a few hand-written workflow scripts, and the GRC tool it already has — and develop the architecture from there. Building the full reference architecture before binding any disciplines would be premature.

Third, the piece is not arguing that the technology stack is the difficult part. The hard work in operationalising PDCA+ is the substrate design itself — choosing the schemas, defining the taxonomies, articulating the translation rules, specifying the measurement constructs, deciding what control objectives the organisation actually operates against. The technology is what carries this work, but the work itself is intellectual and organisational. Pick adequate tools and spend the time on the substrate, not the other way around.

> **THE PRIORITY, IN ONE SENTENCE**
> *The technology is what carries the substrate; the substrate is what makes the framework work. Pick adequate tools, then spend the time on the substrate design — the schemas, taxonomies, translation rules, control objectives, measurement constructs — because the substrate is what determines whether the framework's structural commitments are realised.*

Closing, and what comes next

This piece has named what PDCA+ requires of real infrastructure and how the pieces fit together as a reference architecture. The database holds the substrate's persistent records and the audit trail of its own changes. The workflow engine routes findings based on binding declarations and substrate state. The event bus carries the flow that makes continuous observation possible. The policy engine evaluates rules expressed as code. The data catalog holds the metadata that gives substrate components human meaning. The GRC tool keeps its operational role and emits events that participate in the substrate's flow. Artefact storage holds the evidence the substrate's chain points at.

None of this is exotic. The components are mature, well-documented, broadly adopted, and available from multiple vendors at various price points. What is potentially new is putting them together in service of a substrate that holds the framework's structural commitments — and recognising that the substrate's design, not the technology choices, is what determines whether the operationalisation succeeds.

The South bearing continues from here. The next piece develops code-as-compliance — what it means for compliance rules to be expressed as executable code, where this fits within the infrastructure this piece has named, and how PDCA+ positions itself within the broader movement toward compliance becoming a property of running systems rather than a periodic document review. The infrastructure described in this piece is the substrate code-as-compliance lives in; with the foundations in place, the next piece can develop what is built on top.