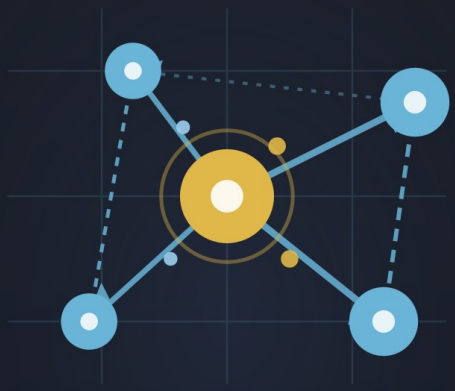
GENERAL REASONING, INC.

MANUAL v1.0



# CHANDRA

## ENTERPRISE

## USER MANUAL

Governed database infrastructure for regulated industries



MANUAL v1.0 · 2026-08-24

Built on the MIT-licensed Chandra Protocol — see Chapter 1

# Chandra Enterprise — User Manual

**Manual v1.0 · 2026-08-24 — Commercial License — General Reasoning, Inc.** *This manual documents Chandra Enterprise, a commercially-licensed product. Chandra Enterprise is built on the Chandra Protocol, which is MIT-licensed — see Chapter 1 for that distinction and Chapter 3 for the full licensing model. Nothing in this manual describes MIT-licensed material unless a chapter says so explicitly. Draft, full manual — assembled from Table of Contents revision 4. Chapters are dated to the source they were verified against; where no verified source exists yet, the chapter says so plainly rather than inventing detail. This is a living document — General Reasoning ships fast, and any chapter can go stale within weeks of being written. Treat a chapter's source note as its expiration warning, not just its citation.*

**A note on manual versioning, since this ecosystem already tracks several version numbers that shouldn't be confused with each other:** "Manual v1.0" above refers to this document only. It is independent of Chandra Protocol's own version (v17.0, per its MIT license banner — Chapter 1), CRC's version (v1.0, coincidentally — Chapter 5), and Chandra Enterprise's own build badge (v31cNNN, incrementing near-continuously — see "Versioning and badge conventions" below). A manual version bump reflects a new pass over this document's content; it says nothing about which CEE build the content was last verified against — that's what each chapter's own source note is for.

## Preface

This manual documents Chandra Protocol, Chandra Enterprise (CEE), and the surrounding standards, tools, and deployment model General Reasoning, Inc. builds on top of them. It is written for the people who will actually operate a Chandra deployment — administrators, integrators, and power users of CEE — not for regulators or examiners (that audience has its own reference at chandrahub.net, see Chapter 10) and not for engineers working on Chandra's own source (that's the codebase's own comments and design docs).

**How this manual is organized.** Part I lays out the concepts you need before anything in Part II will make sense — what Chandra actually is, why it's shaped the way it is, and the standard (CRC) that drives most of its architectural decisions. Part II is operational: how to actually use Chandra Enterprise day to day. Part III covers administration — running the Marshaller, worker fleets, and the pieces still being built.

**Versioning and badge conventions.** Chandra Enterprise's own UI carries a build badge in the form v31cNNN (e.g. v31c425), incrementing with nearly every change — treat it as a precise fingerprint of "what the software could do as of this exact moment," not a semantic version. Chapters in this manual cite the badge (or file version, or fetch date for a live site) they were verified against. If your instance's badge is meaningfully newer than a chapter's citation, verify before trusting operational detail — UI text, field names, and even whole features have moved before between two builds a day apart.

**A note on how this manual is built.** Every operational claim in this manual was checked directly against running code, a live site fetch, or a real data artifact (a personality file, a help-batch export) — not written from general knowledge of what a system "like this" would probably do. Where something is architecture or narrative rather than verified fact, it's attributed to James (General Reasoning's founder) directly, and where something is planned but not yet built, the chapter says so instead of describing

vaporware as shipped. This discipline is itself downstream of Chandra's own governing idea — a system built to make false claims about the past expensive should not be documented with false claims about its present.

---

## Table of Contents



---

# Chapter 1: What Chandra Protocol Is

*Source: chandraprotocol.org (referenced throughout the ecosystem; live-fetched 2026-08-24 for the five-primitives section below); ent-crud.lisp and related schema files (source zip, chandra20260823.zip); chandrahub.net homepage and chandrahub.net/deployments.html (live-fetched, 2026-08-23), the first places the underlying record type is spelled out in plain language for a non-technical audience; chandraprotocol.org/blockchain.html (live-fetched, 2026-08-23) for the blockchain-comparison section below.*

Chandra Protocol is the governance layer underneath everything else in this manual. It is not a product you install by itself so much as a discipline every other Chandra system is built to honor: **every act — every human decision, every agent action — produces a record at the moment it occurs. Immutable. Attributed. Hash-sealed to its predecessor.**


Chandra — The Context Unit Protocol

*The protocol's own official banner (chandraprotocol.org), reproduced here because this is the chapter it actually describes. Note the license line:* ***MIT LICENSE · v17.0 · General Reasoning, Inc.*** *— this applies to Chandra Protocol specifically. It does not apply to Chandra Enterprise, the Marshaller, chandrahub, or anything else this manual documents past this chapter; see Chapter 3 for exactly where that boundary sits.*

## The CU: Chandra's unit of record

The fundamental unit is the **CU** — a **Context Unit** (chandrahub.net is the first source to spell the acronym out plainly; internally and in code comments it's usually just "CU"). A CU is:

**Attested** — it names who or what performed the act (a human, an agent, a service account — see Chapter 5's principal definition, which is Chandra's precise vocabulary for "who")

**Hash-sealed to its predecessor** — each CU's hash incorporates the previous CU's hash, forming a chain. Altering a past CU would break every hash after it, which is what makes the chain tamper-evident rather than merely tamper-logged

**Append-only** — there is no update-in-place operation on a CU once written. A correction is a new CU that supersedes an old one, not an edit to the old one. This is the single most consequential design decision in the whole system, and nearly everything else in this manual traces back to it: Chapter 12's soft-delete/archive/restore lifecycle exists because you can't actually delete a record's history, only its current visibility; Chapter 9's optimistic-locking pattern (loaded_tail_hash) exists because a write has to reference the specific point in the chain it's building on; Chapter 12's Trigger Definitions publish as CUs for the same reason a Form Design publish does — anything that changes system behavior is itself an attested act.

**"The record is not a receipt. It is produced simultaneously with the act it records."** (chandrahub.net's own framing.) This is the distinction worth sitting with: a conventional audit log is written *about* an action, usually by a different system, sometime after the fact, and can in principle be edited or deleted

by whoever has database access to the logging system. A CU is not a log entry describing the action — in Chandra's model, the CU-write *is* the action's completion. Chapter 12's records/create endpoint doesn't "create a record and then log it" — the record's existence and its genesis CU are the same event.

## Chandra's five primitives

Chandra Protocol's own site states plainly that the CU is one of **five primitives**, not the whole story — worth documenting properly rather than letting this manual imply Chandra is just "the CU plus a chain," which undersells two primitives (Snapshot, Tickler) that don't come up naturally anywhere else in this manual and would otherwise go completely undocumented.

**Context Unit** — Atomic, immutable append. One artifact. Full attribution. Always. (This is the CU as described above.)

**Lineage Chain** — Ordered, hash-sealed sequence per governed subject. Tampering is detectable. Broken chains halt new appends. (This is what the rest of this manual informally calls "the chain" or "CU chain" — Lineage Chain is the precise term.)

**Attribution Record** — Every CU carries who acted — human, agent, or system. Anonymous appends are rejected at the protocol level. This is the formal name for the "Attested" property listed above; Chapter 5's principal vocabulary is what populates it.

**Snapshot** — Complete, AI-readable reconstruction of any subject at any point in history. No replay required. Not documented elsewhere in this manual: this implies Chandra doesn't require walking an entire Lineage Chain from genesis to reconstruct a subject's current (or historical) state — a Snapshot is a first-class, directly queryable artifact in its own right, not something every consumer has to derive by replaying every CU. Nothing in the source reviewed so far describes when a Snapshot is generated (on every append? on demand? periodically?) — that's a real gap worth closing once there's source to verify against.

**Tickler** — A forward-scheduled CU with a trigger time and a target. Described as "the federation primitive." Cancellation is a superseding append, not a deletion — consistent with the append-only discipline above (Chapter 1's "correction is a new CU, not an edit" rule applies to scheduled future actions too, not just past ones). Also not documented elsewhere in this manual, and its description as "the federation primitive" suggests a connection to Chapter 10's chandrahub/forest concept and the "Cross-Forest Trust Protocol" referenced as forthcoming on chandrahub.net's Standards page — but that connection is speculative, not confirmed, and shouldn't be treated as established until there's real source describing how Ticklers actually cross a forest boundary.

Primitives 4 and 5 (Snapshot, Tickler) are genuinely new territory for this manual — nothing elsewhere here describes either one operationally. Treat them as named-and-defined but not yet operationally documented, the same honesty standard the rest of this manual applies to unbuilt features — the difference here is these primitives are stated to exist at the protocol level already, just not yet walked through in this manual.

## Why append-only is the whole point

An append-only chain converts a hard question — "can I trust this history?" — into an easy one — "does this hash chain verify?" You don't have to trust the operator's honesty going forward; you can check the math. This is what lets Chapter 10's audience (examiners, auditors) verify a GABA cert or trace an audit opinion back to its work papers without taking anyone's word for it, and it's what makes Chapter 9's migration-verification pattern (checking chain/index after a write, not just trusting the HTTP 200) a meaningful practice rather than theater.

It also explains a few things that might otherwise look like limitations elsewhere in this manual: - Why CSV import (Chapter 12) is add-only with no update/upsert path — an "update" that silently overwrote history would be exactly the kind of edit-in-place the whole chain exists to prevent. A correction goes through the same append-only records/update call as any other change, producing its own CU. - Why a locked/published Calculated Field formula (Chapter 12) or Form Design can't be edited in place — the discipline of "retire, don't rewrite" is Chandra Protocol's append-only principle applied to schema, not just data.

## What governance means concretely

"Governed" is used loosely across the industry to mean almost anything about oversight. In Chandra's vocabulary it means something specific: **every mutation is an attested, chained fact**, full stop — not "most mutations get logged," not "mutations above a certain sensitivity get reviewed." Chapter 9 documents the one real gap in this story as it stands today: request-level authentication and access control are not visible anywhere in the current source (confirmed by direct search of the source zip — no bearer tokens, no API keys, no auth middleware). Chandra Protocol governs *what happened*, completely and verifiably; it does not yet, on its own, govern *who was allowed to make it happen* at the network-request level. That's a real, open architectural gap worth naming rather than glossing over, and it's presumably part of what AegisGenera (Chapter 19) and Chandra Passport (Chapter 15) are meant to close once they exist.

## Why Chandra is not a blockchain

This is worth addressing directly because the word "immutable" reliably makes people reach for the wrong comparison. Chandra Protocol's own site (chandraprotocol.org/blockchain.html) states the distinction more precisely than this manual could paraphrase it, so most of what follows tracks that page closely.

**Blockchain and Chandra use the same primitive — a hash chain — to solve different problems.** Blockchain answers *how do mutually distrustful parties agree on a shared ordered ledger without a central authority?* That's a real problem, but it isn't a regulated enterprise's problem. Chandra answers *how does an organization prove what a governed human, agent, system, or process did, under what authority, at the moment it happened?* A regulated firm already has an accountable, named institution — it doesn't need to eliminate trust in that institution, it needs to make that institution's own actions provable. As the source puts it: *"Blockchain tries to eliminate the need to trust an institution. Chandra makes the institution's actions provable."*

Concretely, this means Chandra has no need for miners, validators, staking, gas, mempools, or global transaction finality — none of blockchain's adversarial-consensus machinery. It needs the opposite set of properties: identity binding, authorization evidence, schema conformance, role/scope capture, policy references, contemporaneous event production, hash continuity, and examiner-readable reconstruction. Blockchain's ideal is that no central authority can control the ledger; Chandra's ideal is nearly the inverse — every authority is explicit, attributed, bounded, and auditable. A regulator does not want to hear "the decentralized network did it" — they want to know which person, system, policy, or approval path caused an outcome, and Chandra is built to answer exactly that question.

**Topology reflects this difference directly.** A blockchain's vocabulary is *network → nodes → blocks → transactions*. Chandra's vocabulary is *marshaller → instance → domain → hub → spoke → context unit* (Chapters 2, 4, and 7 cover instance/hub/spoke/marshaller in detail; **domain** as a distinct governance boundary — e.g., an FDA design-control spoke versus a FINRA supervision spoke being different regulatory realities, not just different transaction streams — is a term worth tracking as this manual's chapters on the Spine and Hub/Spoke model are revised). An auditor essentially never asks "show me the entire global ledger" the way a blockchain explorer would answer; they ask "show me the record for this control, this exception, this authorization" — which is exactly what a Spoke's own reconstructible CU chain (Chapter 1, above) is built to answer directly, without reconstructing anything from a global transaction stream.

**External anchoring is optional, not constitutive.** Chandra chains can optionally anchor a root hash or daily digest to an external ledger, which makes later tampering harder to deny — but that's supplementary evidence, not the governance mechanism itself. Chandra may anchor to external ledgers; it does not outsource governance to them. A blockchain can prove a hash existed at a point in time. Chandra proves what that hash *meant* inside a governed system — who acted, under what authority, against what policy, with what evidence.

This chapter covers the protocol layer only — the CU, the chain, the attestation principle. Chapter 2 covers Chandra Enterprise, the product built on top of it (Hubs, Spokes, forms, browses). Chapter 5 covers CRC, an independent standard that happens to be the design philosophy behind *why* Chandra's deployments look the way they do, but which is not itself part of the protocol. Don't conflate the three — Chandra Protocol is the ledger; CEE is the application; CRC is the architectural doctrine that says how few, how isolated, and how closed the *deployment* of that application should be.

---

# Chapter 2: What Chandra Enterprise Is

*Source: chandra-enterprise-editor.html (build v31c425+); ent-hub.lisp, ent-tree.lisp, ent-spine.lisp (source zip); vatican.chandrahub.net (live-fetched, confirms the environment-root model described below).*
Chandra Enterprise (CEE) is the application layer built on Chandra Protocol. If Chapter 1's protocol is the ledger, CEE is the database-plus-UI that regulated organizations actually work in day to day — and it is deliberately shaped like a FileMaker-generation business database (Chapter 12 documents the operational parity directly: Sort, Found-Set operations, Found navigation, Duplicate, Import/Export, Portals) rather than a bespoke enterprise app, because that's a UI paradigm regulated-industry staff already know.

## The data model: Hub and Spoke

CEE's schema vocabulary is **Hub** and **Spoke**:
A **Hub** is a governed table — a business object type (e.g., "Loan Applications," "Vendor Contracts," "Holy See Apostolic Acts" — see Chapter 6's Vatican personality example). A Hub carries a **Form Design** (Chapter 14): the field schema, validation rules, and layout that every record in that Hub conforms to.
A **Spoke** is a governed row — one record inside a Hub, with its own CU chain (Chapter 1), its own lifecycle state (active/archived/deleted — Chapter 12), and, if it's a child record, a parent Spoke it's attached to (the mechanism behind Chapter 12's Related-Record Portals).
This is a deliberately narrow vocabulary. ent-environment-roots.lisp reserves a small, closed set of Spoke *types* — hub-info, form-design, help-context, bug-report, attachment, integration-genesis — meaning even CEE's own internal machinery (its help content, its bug-report system, its file attachments) is represented as governed Spokes inside the same Hub/Spoke model business data uses, not as a separate privileged internal data layer. There is no backstage.

## Environment roots: Acceptance and Production

Every CEE spine has (at minimum) two environment roots: **Acceptance** and **Production** (ent-environment-roots.lisp — *ent-environment-roots* is literally '("acceptance" "production")). This is confirmed live on vatican.chandrahub.net, whose UI shows a SPINE / CANARY / PRODUCTION toggle. Acceptance is where a spine configuration is tested; a distinct "Accept Acceptance Spine" action (seen live) commits the current test spine as the accepted source for Production — the confirmation dialog is explicit that this only sets up what Production *should* look like, and that "Production creation or any later Production change remains governed by the Production/Concordia rules," i.e., promoting to Production is its own separately-governed act, not a side effect of accepting an Acceptance spine.
Help and Bug Reports are **root-scoped** — each environment root gets its own Help/Bug-Report Hubs, per the boot-log comment in start-enterprise.lisp: *"environment roots — Acceptance/Production anchors with root-scoped Help and Bug Reports Hubs."* This is a smaller, earlier instance of the same isolation instinct documented at length in Chapter 18: don't let content homed in one governance context bleed into another, even within a single spine.

## What CEE adds on top of the protocol

Chapter 1 draws the line already, but concretely, CEE is what turns "an append-only attested chain" into something a compliance analyst can actually use:

**Forms** (Chapter 14) — schema, field types, validation, and now (as of build v31c425) calculated fields (Chapter 12 §8)
**Browses** — the found-set/sort/navigation toolbar documented exhaustively in Chapter 12, which is CEE's entire day-to-day surface for a working analyst
**Reports and exports** — CSV and, as of the same build, PDF generation (Chapter 12 §9)
**Governed automation** — Trigger Definitions (Chapter 12 §10), which is how CEE does auto-entry and cross-field validation logic without ever exposing a free-form scripting surface (the save path explicitly pattern-matches and rejects anything that looks like injected code)
**Related-table modeling** — sub-Hubs (Chapter 12 §7), created through a governed /api/enterprise/sub-hub/create call, not a schema-migration side channel
None of this exists at the protocol layer. Chandra Protocol doesn't know what a "Form Design" is; CEE is the layer that decided a Hub needs one, and that a form's publish should itself be a CU.

## CEE is not the only thing built on the protocol

It's worth being explicit that CEE is *a* product on Chandra Protocol, not *the* product. The Marshaller (Chapter 7) and Chandrahub (Chapter 10) are both separate applications sitting on the same protocol, with their own Hub/Spoke-shaped internal data (per Chapter 18's Help-content architecture) but serving entirely different jobs — fleet discovery and public governance reference, respectively, rather than line-of-business record-keeping. When this manual says "governed," it means governed the Chapter 1 way, regardless of which of these three products is doing the governing.

# Chapter 3: Licensing Model

*Source: chandra-banner.svg (Chandra Protocol's own official banner graphic, MIT LICENSE · v17.0 · General Reasoning, Inc. — the clearest first-party confirmation of the protocol's license and version seen so far); crcstandard.org homepage (live-fetched — "MIT LICENSE · v1.0 · GENERAL REASONING, INC."); chandrahub.net homepage (live-fetched); James, directly, confirming the split in exactly these terms: "Core is MIT. The enterprise version requires a commercial license."*

Chandra's licensing follows a deliberate split, confirmed directly rather than inferred: **the core (Chandra Protocol) is MIT-licensed; Chandra Enterprise requires a commercial license.**

## What's open (MIT-licensed)

**CRC** (Chapter 5) — "CRC is published as an open standard… MIT License · v1.0 · General Reasoning, Inc." Anyone can read the standard, score their own stack against it, and implement it without a license from General Reasoning. The stated intent, in the standard's own words: *"The goal is not General Reasoning's business — it is a regulated infrastructure posture that can actually survive the adversary that now exists."*

**Chandra Protocol** (Chapter 1) — MIT-licensed, confirmed directly on the protocol's own official banner (**MIT LICENSE · v17.0 · General Reasoning, Inc.**), and referenced as an open protocol throughout chandrahub.net's own framing ("Chandra is an open protocol for append-only, hash-chained, attributed audit records"). The protocol's data shape and governance principle are meant to be implementable by anyone, the same way CRC's scoring framework is. Note the version numbers are tracked independently — the protocol itself is at v17.0 as of this banner, while CRC (a separate, related standard) is at v1.0.

## What's commercial

**Chandra Enterprise requires a commercial license, stated directly.** Chandra Enterprise (Chapter 2) and the surrounding operational tooling (Marshaller, Spine Editor, worker orchestration) are General Reasoning's proprietary product — this is the thing sold to compliance buyers, not given away under the same MIT terms as the core protocol it's built on. The distinction mirrors a common and well-understood pattern: the *protocol* is open so the industry can adopt its data shape and governance vocabulary without a gatekeeper, while the *product that makes running it easy* is the actual commercial offering.

## What this split means practically

**If you're deploying Chandra as a client:** you are licensing Chandra Enterprise (and, per Chapter 10, ideally running it as your own sovereign deployment rather than depending on a shared instance) — but you are never locked into a proprietary *audit format*. Because CU structure and CRC scoring are open, an examiner (Chapter 10's audience) or a competing vendor can verify your compliance posture without needing a license from General Reasoning to understand what they're looking at. This is a meaningful claim precisely because it's unusual: most compliance tooling vendors have some incentive to make their own attestation format at least mildly proprietary, since that increases switching cost. CRC and Chandra Protocol being MIT-licensed removes that incentive structurally — General Reasoning's business model apparently depends on being the best implementation of an open standard, not on owning the standard itself.

**If you're building on top of Chandra** (Chapter 9's integration audience): the wire protocol you're calling is part of the commercial CEE product, not the open standard — Chapter 9's endpoint reference

documents an interface to a licensed product, not a public API in the way CRC's scoring criteria are public. Nothing described in Chapter 9 is available royalty-free just because CRC is.

**Open item for this chapter:** the exact legal terms of CEE's commercial license (per-seat, per-instance, per-organization, usage-based) haven't been sourced from anything reviewed so far. This section should be expanded with real license text or pricing-page content once that exists, rather than left to speculation.

---

# Chapter 4: The Spine

*Source: index.html (Industry Configurator); personality-vatican-8level.json (real 8-level Holy See/Vatican personality artifact); personality-general_reasoning-finra-1783102313180.json; ent-spine.lisp, ent-spine-validation.lisp, ent-tree.lisp (source zip); vatican.chandrahub.net (live-fetched, confirms the tree UI).*
The **Spine** is Chandra's model of an organization's own regulatory or operational hierarchy — the scaffolding that determines what a "Level" means for a given deployment, before any actual data (Hub/Spoke, Chapter 2) is attached to it.

## Levels, not tables

A Spine is an ordered sequence of **Levels** — Level 1 through Level N — each with:
A **label** (e.g., "Holy See / Apostolic See," confirmed from the real Vatican personality)
A **parent Level** (or none, for the top of the spine) and a list of **child Levels**
A **level form design reference and context** — metadata describing that Level's own identity (canonical label, internal term, description, primary authority, typical records — all confirmed fields from the Vatican artifact)
An **attestation boundary flag** — whether this Level is itself a point where governance/attestation is formally anchored (the Vatican artifact's top level, Holy See/Apostolic See, is flagged attestation_boundary: true)
Critically, a Level is **not** a Hub. The real personality artifacts are explicit and repeated on this point: Hub/table authoring, Spoke/row authoring, and physical instance assignment for a given Level are all "deferred_to_spine_editor" — a Level is a slot in the organizational hierarchy; what tables and records actually live inside it is decided later, by a different tool (Chapter 13), not by whoever designed the Spine's shape.

## Numbering: stored bottom-up, displayed top-down

This is a real, easy-to-get-wrong detail, confirmed directly from the Vatican artifact's own logical_spine metadata:
"numbering_starts_at": 1,
"display_order": "top_to_bottom",
"stored_numbering": "bottom_up",
"invariant": "Level numbers are logical industry hierarchy coordinates only. They are not physical Instance numbers."
Concretely: in the Vatican 8-level spine, Level 8 is "Holy See / Apostolic See" — the *top* of the organizational hierarchy — while numbering starts at 1 at the bottom. When the Spine is displayed, it renders top-to-bottom (Holy See at the top of the screen), but the stored level number counts up from the bottom. **Do not assume Level 1 is the top of a spine, and do not assume a Level's number has any relationship to which physical worker instance eventually hosts it** — the artifact's own invariant states plainly that these are logical hierarchy coordinates, not physical instance numbers. Physical instance assignment is Chapter 13's job, computed later, independently of how the levels happen to be numbered.

| | | |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | **How a Spine differs from a Hub/Spoke schema** |

It's worth stating the distinction plainly, since both are hierarchical and it's easy to conflate them:

**Spine / Level**
**Hub / Spoke**
What it represents
Organizational or regulatory structure
Actual business data
Who defines it
Industry Configurator (Chapter 6) produces the logical shape; Spine Editor (Chapter 13) authors it directly and packs it physically
Form Design (Chapter 14) per Hub
Governance unit
A Level can be an attestation boundary
Every Spoke has its own CU chain (Chapter 1)
Physical mapping
One or more Levels get packed into a physical worker instance (Chapter 13, using the CRC/ISS formula from Chapter 5)
A Hub lives inside whichever instance hosts its owning Level
A useful mental model: the Spine answers "what kind of organization is this, and how is its authority structured" (a Vatican deployment's Spine looks nothing like a broker-dealer's); Hub/Spoke answers "what data do we actually keep, and in what shape" within whatever Level that data belongs to.

## Spine validation

ent-spine-validation.lisp enforces structural rules on a Spine before it can be accepted (Chapter 2's Acceptance/Production distinction is exactly where this validation gets exercised): level numbers must be contiguous, exactly one Level has no parent (the top of the hierarchy, regardless of what its *number* happens to be under the bottom-up storage rule above), and parent/child references must resolve to Levels that actually exist in the same Spine. The Industry Configurator (Chapter 6) runs a version of this same validation live as you edit — it's the mechanism that keeps a hand-edited Level spine internally consistent before it's ever exported as a personality artifact.

# Chapter 5: The CRC Standard and Isolation Surface Size

*Source: crcstandard.org homepage and /scoring.html (live-fetched, 2026-08-23); isolation-surface.html (crcstandard.org, uploaded copy); personality-vatican-8level.json's crc_isolation_policy block. CRC is independent of Chandra — it's an open standard General Reasoning publishes, MIT-licensed v1.0, that happens to be the design philosophy behind why Chandra deployments look the way they do.*

CRC — **Consolidate, Reduce, Close, Boundary** — is General Reasoning's published Minimum Surface Standard for AI-era regulated infrastructure. Understanding it is a prerequisite for the rest of this manual's architecture chapters: the Spine (Chapter 4), the Industry Configurator's scope boundary (Chapter 6), the Spine Editor's actual job (Chapter 13), and the entire per-surface isolation posture in Chapters 10 and 18 are all downstream of this standard's argument.

## The Dunbar Perimeter

CRC's argument starts from a claim about human cognition, not computers. Anthropologist Robin Dunbar observed that humans can maintain stable relationships with roughly 150 people before cognitive load becomes unmanageable. CRC extends this to systems: a CISO can hold one application's architecture in their head with clarity, maybe two — by five or ten interconnected platforms, they're past what human cognition can manage.

So organizations partitioned by team: the Salesforce team secured Salesforce, the SAP team secured SAP, the identity team secured Active Directory. Each team felt like a perimeter. **CRC calls this the Dunbar Perimeter** — a security boundary created not by architecture, but by the cognitive limit of the humans responsible for it. It was never a real control plane; the systems were always connected, the connections just weren't fully seen.

**The SaaS business model industrialized this.** Every vendor sells a best-of-breed, independently-certified, independently-secured solution for one domain — and every one of them connects to all the others. Each integration, each SSO connection, each shared data flow crosses a Dunbar Perimeter and is secured by no single owner. Individually secure SaaS systems can compose into a collectively insecure architecture.

This was survivable as long as the attacker had a Dunbar limit too — a human attacker, even a sophisticated one, could only hold so many systems in their head at once, and picked targets accordingly. **CRC's stated reason this no longer holds is Anthropic's Mythos model:** a model with no Dunbar limit, that doesn't partition systems into cognitive domains, doesn't fatigue, and doesn't privilege one context over another — it constructs and traverses the full dependency graph across identity systems, SaaS platforms, and data flows simultaneously and autonomously. The connections that were always there, running through every integration boundary that nobody owned, become first-class attack paths the instant something can see the whole graph at once. **"If Mythos cannot reach it, Mythos cannot chain it"** is CRC's stated design objective, and it's the reasoning behind the isolation choices documented throughout this manual — including, per James directly, why system Help content itself is split by authority rather than kept in one store (Chapter 18): a monolithic store is a graph node, and a graph node is something Mythos can reach.

## The Governed Server Boundary Prerequisite

Before any of the four pillars apply: **CRC assessment applies exclusively to governed server deployments under formal organizational control.** End-user devices, developer workstations, and any environment where AI agents with computer-use capability operate outside a governed boundary are out of scope. If that prerequisite isn't met, a CRC score is not zero — **it is inapplicable.** This is a scope gate, not a caveat to note in passing; a deployment that fails the prerequisite isn't "scoring poorly on CRC," it's not a CRC-assessable system at all until the boundary condition is fixed.

## The four pillars

CRC scores a deployment 0–4 on each of four pillars, for a total of **0–16**. **Regulated deployment certification requires 13 or higher.**
**01 Consolidate** — *How many integration boundaries exist between regulated-workflow apps?*

0 (Uninspected): Multiple apps, boundaries unowned, uninventoried
1 (Inventoried): Boundaries documented, ownership assigned, data flows mapped
2 (Controlled): Boundaries owned, monitored, traffic logged, API contracts enforced
3 (Consolidated): Majority of workflow on one governed platform, remaining boundaries are conscious decisions
4 (Minimal): Single governed platform, zero unowned boundaries, every data flow is an immutable attributed audit record
**02 Reduce** — *What's in the production execution environment beyond the app itself?*
0 (Uninspected): General-purpose OS, full package universe, shell accessible
1 (Inventoried): Services documented and pruned, OS hardened to a benchmark (CIS/STIG), shell restricted and logged
2 (Controlled): Containerized, no package manager in production, break-glass-only shell
3 (Reduced): Purpose-built OS image, no shell binary present, read-only root filesystem, signed boot chain
4 (Minimal): TPM-attested measured boot, immutable filesystem, the application *is* the OS
**03 Close** — *What can reach production from outside the governed boundary?*
0 (Uninspected): Public-facing services, unrestricted egress, ports open by default
1 (Inventoried): Firewall rules documented, egress filtered by category, topology mapped
2 (Controlled): Ingress restricted to known authenticated sources, egress allow-listed, zero-trust enforced
3 (Closed): Circular topology, nodes only talk to other known authenticated nodes, mutual TLS, no public-facing endpoints
4 (Minimal): Single auditable egress point, certificate-pinned endpoints only, every crossing logged as a first-class audit event
**04 Boundary** — *How completely are external AI inference endpoints and computer-use agent access governed?*
0 (Uninspected): Unrestricted external AI API calls, no logging, no endpoint registry
1 (Inventoried): Endpoints documented, agent access identified, no formal controls beyond documentation
2 (Controlled): Rate-limited, logged, response-validated, no unmanaged computer-use access
3 (Hardened): Cert-pinned per endpoint, request signing, anomaly detection, circuit breakers, sanitized payloads
4 (Attested): All hardening active, **GABA attestation complete for every endpoint** (Chapter 11), residual risks formally signed by an authorized signatory

A stale artifact worth knowing about: the scoring page's own meta-description still reads "Score 0-12 across three pillars" — a leftover from before Pillar 04 (Boundary) was added. The live scoring UI itself correctly reflects four pillars and a 0–16 scale; don't repeat the stale three-pillar framing if you're citing this standard elsewhere.

## General Reasoning's own stack, mapped to the pillars

CRC's own homepage names how General Reasoning's products satisfy each pillar, which is worth reproducing here since it directly explains why this manual's later chapters exist:
**Consolidate** → **DXMachine** brings regulated workflows that would otherwise span Salesforce, SAP, Workday, ServiceNow, and integration layers onto one governed platform, eliminating the nodes and edges between them.
**Reduce / Close** → "Purpose-built infrastructure running **Chandra Protocol** and **DXMachine** — nothing else." A purpose-built OS running exactly one thing, with a circular network topology and a single certificate-pinned egress point.
**Boundary** → every external AI inference endpoint named, certificate-pinned, logged, and formally attested; computer-use agents require explicit **GABA** attestation (Chapter 11) separate from inference-only endpoints. This is very likely AegisGenera's (Chapter 19) job once it's built, based on the standard's own description of what Pillar 04 hardening requires — cert-pinned endpoints, request signing, a governed endpoint registry — but that connection hasn't been confirmed against AegisGenera's own source, since none exists yet.

## Isolation Surface Size (ISS): Pillar 01, made quantitative

Attack surface and isolation surface are different questions. Attack surface asks *where can an intruder get in*. Isolation surface asks *how much sits behind one boundary once they're in* — a master key protecting more doors isn't a busier key, it's a more valuable one to steal.
**Cost as a function of surface size:**

$$\text{Cost}(S) = p \cdot \bar{v} \cdot S \cdot T + (c_r + c_c \cdot f) \cdot (T/S)$$

**Optimal surface size:**

| | |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | $S^* = \sqrt{[(c_r + c_c \cdot f) / (p \cdot \bar{v})]}$ |

**Symbol**
**Meaning**
S
Isolation surface size — principals per certificate/boundary
T
Total principal population across the deployment
p
Per-principal compromise probability
$\bar{v}$

Average privilege/value per principal
c_r
Fixed cost of issuing and maintaining one boundary
c_c
Convenience cost per cross-boundary workflow
f
Fraction of workflows that cross a boundary
The two terms trade off: safety cost rises with surface size (a bigger boundary is more valuable to compromise), while complexity/convenience cost falls (fewer, larger boundaries mean less provisioning overhead and fewer workflows hitting a seam). The curve has an interior minimum — that minimum is $S^*$.

**The key result: T cancels out of $S^*$.** Optimal surface size is population-independent. A fifty-person team and a fifty-thousand-person enterprise should target the *same* $S^*$ — the enterprise just needs more boundaries at that size, not bigger ones. This is the mathematical justification behind Chapter 13's per-Level physical-instance packing: you don't scale a deployment by making one worker instance handle more; you scale it by adding more instances at the same optimal size.

**Worked example (ServiceNow + Salesforce, shared SSO domain):** T=500, p=0.008, $\bar{v}$=25, c_r=60, c_c=30, f=0.35.

$S^* = \sqrt{[(60 + 30 \cdot 0.35) / (0.008 \cdot 25)]} = \sqrt{352.5} \approx 19$
Optimal: ~19 principals per boundary → ~26 boundaries needed. Current: 500 principals under one shared SSO boundary → the surface is roughly **26× larger than optimal**, and total cost runs roughly 4.5× above the curve's minimum. The formula doesn't say the configuration is inoperable — it says it's carrying materially more risk-adjusted cost than the same environment, properly partitioned, would require.
**The precise definition of principal**, confirmed verbatim from a real personality artifact's schema and worth using consistently everywhere this manual talks about "who" governs or accesses something: *"a distinct authority-bearing signing credential: human, agent, service account, API key, examiner, workflow engine, or integration identity that can authorize, attest, append, route, or access privileged Chandra content."*

## Where the formula travels, and where it's actually computed

This is a detail easy to get backwards. The Industry Configurator (Chapter 6) carries the ISS formula in every personality artifact it produces, under a crc_isolation_policy block — but its own configurator_role field is explicitly "not_computed_here". The Configurator hands the formula and its principal definition forward as *policy*; it never runs the calculation itself. The tool that actually computes $S^*$ against real deployment inputs and packs logical Levels (Chapter 4) into physical worker instances (Chapter 8) is the **Spine Editor** — see Chapter 13.

## Open items for this chapter

attack-chain.html and gabastandard.com (referenced in CRC's own nav bar) have not yet been reviewed in depth — the former likely elaborates Pillars 02/03, and the latter is Chapter 11's subject. Both should be folded in once reviewed rather than assumed from the fragments seen so far.

# Chapter 6: The Industry Configurator

*Source: index.html (Industry Configurator source, including the crcPolicy() and buildPersonality() functions); three real personality artifacts examined directly — personality-general_reasoning-finra-1783102313180.json, personality-general-reasoning-sifi-1781207518833.json, and personality-vatican-8level.json.*

The Industry Configurator produces a **personality** — a portable artifact describing a regulatory domain's logical organizational structure, ready to seed a new Chandra Enterprise deployment. It has been run for at least FINRA, a "SIFI" (systemically important financial institution) domain, and — as a genuine stress test of how general the model is — an 8-level Holy See/Vatican City State spine (domain_class: "vatican", industry: "Holy See / Vatican City State — Sui Generis").

**The live tool is at [config.genreason.com](config.genreason.com).** This is the same instance referenced throughout chandrahub.net's own beta-onboarding flow (Chapter 10) — generate a personality here, then email the resulting JSON to inquiries@genreason.com to actually get a deployment stood up.

## What a personality contains

Every personality artifact examined shares the same top-level shape: artifact_kind, standard (a versioned schema name, e.g. "chandra-industry-level-personality-v1"), industry, domain_class, organization, submitter identity, an architecture_contract, logical_level_count, a logical_spine (Chapter 4's Level structure), a crc_isolation_policy block (Chapter 5's formula, carried as policy), a list of fields excluded_from_personality, and a validation result.

## The Configurator's scope is explicitly, deliberately narrow

This is the single most important thing to understand about this tool, and it's stated in its own source rather than inferred: the Configurator's architecture_contract names its scope as *"map and edit ordered industry Level spine"* — nothing more — and explicitly lists forbidden_outputs:
instance_count, tier_count, deployment_bindings, hub_templates,
spoke_templates, hubs, spokes, ports, urls, runtime_health
Every one of these is explicitly deferred_to_spine_editor, alongside Hub/table definitions per Level, Spoke/row definitions per Hub, form designs, and CRC physical-instance packing. The Configurator's own excluded_from_personality list reinforces this from the other direction: Hub/table schema, Spoke/row schema, physical instance count, instance addresses, ports, localhost URLs, deployment hostnames, runtime health state, and environment-specific routing are all named as things a personality artifact will never contain.

**Why this boundary exists, and why it matters enough to be enforced in code rather than just convention:** a personality is meant to be a portable description of an industry's organizational logic — the same FINRA personality should describe what a FINRA-regulated broker-dealer's spine looks like regardless of whether that specific deployment ends up running on one worker instance or twenty, on one Linode box or a client's own AegisGenera hardware (Chapter 10). If the Configurator baked in instance_count or ports, every personality would be tied to one specific physical deployment the moment it was generated, defeating the point of having a reusable, industry-general artifact at all. The boundary is also a direct expression of Chapter 5's Consolidate/Reduce discipline applied to the tooling itself: the Configurator doesn't know about physical infrastructure because knowing about physical

infrastructure isn't its job, and a tool that does one job with a hard boundary is easier to reason about and secure than one that does several.

## Live ISS scoring during configuration

As domain, organization size, and workflow selections are made in the Configurator, the tool scores the resulting configuration against Chapter 5's ISS formula live — visible directly in the source as the crcPolicy() function, which is called at multiple points while building the personality (crc_isolation_policy: crcPolicy()). Note again the configurator_role: 'not_computed_here' field embedded in that same function's output: even while giving you live feedback as you configure, the Configurator is not the system of record for the actual physical-instance math — that calculation belongs to the Spine Editor (Chapter 13), which consumes the same formula and principal definition the Configurator hands it, against real deployment-specific complexity/risk inputs the Configurator never sees.

## A worked example: the Vatican personality

The Holy See/Vatican personality is worth walking through briefly because it demonstrates the model handling something far outside the FINRA/CMMC/HIPAA-style domains this tooling was probably first built for. Its 8-level spine runs from Level 8 (Holy See / Apostolic See — the top of the hierarchy, per Chapter 4's stored-bottom-up/displayed-top-down numbering) down through the sovereign structure of the Holy See, with each Level's level_form_values carrying domain-appropriate metadata: canonical_label, internal_term, description, primary_authority (e.g., "Roman Pontiff / Secretariat of State" for the top Level), and typical_records (e.g., "Apostolic constitution, motu proprio, Lateran Treaty file"). The Level 8 entry is flagged attestation_boundary: true — the Configurator's schema supports marking specific Levels as formal governance anchors, not just organizational nodes, which is presumably relevant to how the Spine Editor eventually treats that Level when computing isolation surfaces.

The fact that the same underlying tool, schema, and CRC isolation policy produce a coherent personality for a broker-dealer's FINRA obligations and for the Holy See's sovereign governance structure is a reasonable proof point for how domain-general the Level/Hub/Spoke model actually is — worth keeping in mind when Chapter 4 talks about a Spine as a general organizational-hierarchy concept rather than something scoped narrowly to financial regulation.

## How a personality becomes a deployment

The pipeline, confirmed end-to-end across the Configurator source, the personality schema, and Chapter 5's formula:

**Industry Configurator** (this chapter) → produces a personality: logical spine, industry vocabulary, per-Level metadata, zero physical-deployment opinions.

**Spine Editor** (Chapter 13) → consumes the personality, authors the actual Hub/Spoke schema per Level, and computes physical instance count / packs Levels into instances using the CRC/ISS formula the personality carried forward as policy.

**Worker instances** (Chapter 8) → the realized, physically isolated surfaces that result.

This chapter documents step 1 only. Don't describe the Configurator as doing anything past producing the personality artifact — that boundary is enforced in its own code, not just a convention this manual is choosing to respect.

# Chapter 7: The Marshaller

*Source: chandra-marshaller.html (UI, build v37, 2026-08-23); chandra-marshaller-attestation-console.html (companion tool, v18); ent-marshaller.lisp (source zip); marshaller.chandrahub.net (live-fetched, 2026-08-23).*
The Marshaller is Chandra's organization-and-spine directory — a fleet-level registry that tracks *where* governed deployments are, without itself holding any of their governed data. Its own file header states its scope precisely: **"organization → existing spine address registry."**

## Address-only persistence: what Marshaller actually stores

This is the architecturally load-bearing fact about the Marshaller, confirmed both in the live UI copy and in ent-marshaller.lisp's own design history: **the spine address is the only thing Marshaller persists.** The UI states this to the operator directly: *"This is the only value Marshaller persists. VERIFY reads the responding spine live; the result is never saved here."*
Everything else shown about a registered organization — instance name, build version, environment, personality ID, personality file, instance type — is a **live read**, fetched at VERIFY time from that spine's own /api/enterprise/self manifest, and explicitly discarded rather than cached. The UI's own field-level help spells out the consequence directly: *"Read-only live result from the spine. Marshaller discards it and can reconstruct it again from this address."*

**Why this matters, and a real gotcha documented in the source history:** because VERIFY is a point-in-time snapshot and not a subscription, a Marshaller org-detail view can go *stale relative to the spine's current state* — if a personality gets set directly on the spine after the last VERIFY, Marshaller's displayed view won't reflect that until VERIFY is run again. ent-marshaller.lisp's own v9 changelog entry addresses this directly: *"Marshaller's org detail view was stale relative to a spine's current state (VERIFY is a point-in-time snapshot; re-running it is the only way to pick up a personality set directly on the spine afterward — not a bug, just how VERIFY works)."* Treat a Marshaller view as "as of the last VERIFY," not as continuously live, and re-run VERIFY before trusting a detail that might have changed on the spine side.

The v9 changelog entry also confirms the *breadth* of what a VERIFY captures got fixed recently: earlier versions only captured a hand-picked subset of /api/enterprise/self fields; as of v9, the entire self response is embedded verbatim as one observed_self_json blob rather than requiring a new individually-named field every time self grows a key.

## GR Marshaller vs. Client Marshaller

Two deployments of the same tool serve different scopes:
**GR Marshaller** — General Reasoning's own fleet directory, tracking every organization GR has a relationship with. Live at marshaller.chandrahub.net, confirmed by direct fetch: its Help Index is explicitly labeled "LOCAL MARSHALLER ACACHE" in the UI (see Chapter 18), and its own marshaller-overview help context states its isolation from other surfaces directly: *"Marshaller is the fleet-level discovery and observation surface. Its persistent registry authority is local to the Marshaller ACACHE. It does not read or write CEE or worker Help Contexts."*

**Client Marshaller** — the same tool, self-hosted by a client, tracking that client's own fleet of spines rather than GR's book of business. Nothing in the source distinguishes these as different code paths — it's the same product, deployed with a different (and much narrower) scope of organizations to track.

## Registering and verifying an organization

The Marshaller workflow, per the live UI:
**Add Spine Root** (or **Import** — see below) — enter an organization name, a spine address, an optional Markdown description, a status (active/inactive/unverified/retired), and optional operator notes.
**VERIFY** — reads the responding spine live at that address and displays its manifest identity. Successfully verifying sets the registration's status to active automatically. There is no separate "Retire" action anymore — setting status to retired and saving takes the organization out of service the same way any other field edit does, and (per Chapter 1's governance principle) that save still appends a CU like any other edit.
**Bulk import** — one organization per line, tab-separated: Name, Spine address, and an optional Notes column. Re-running an import is explicitly safe: a name that already exists is updated, not duplicated, matching the regular single-entry save form's behavior. An optional toggle also runs VERIFY on each imported address automatically (fills in Type/Personality), at the cost of being slower and only working for addresses that are actually reachable at import time.

## What Marshaller deliberately does not do

Marshaller does not proxy, mirror, or cache business data from the spines it tracks — it has no Hub/Spoke business records of its own beyond its own address registry, help content, and bug reports. It is a discovery and observation layer, not an aggregation layer: if you need to actually query or modify data on a registered spine, you go to that spine's own CEE or API surface (Chapter 9) directly. Marshaller's job ends at "here is this organization's current address and what its manifest says right now."

## The Attestation Console

chandra-marshaller-attestation-console.html (v18, badge v10 in its own footer) is a companion tool: **"API-backed registry telemetry"** with an explicit caveat baked into its footer copy — *"geographic observations are approximate and non-attested."* This distinction is worth taking at face value: whatever this console shows is *not itself* a CU-backed attested fact the way a spine's own records are (Chapter 1) — it's observational telemetry about the fleet, presented with an honest disclaimer about its own limits rather than dressed up as governed data it isn't. Its precise relationship to the chandrahub.net attestation/GABA-cert narrative (Chapter 10) has not yet been confirmed directly and should be verified with James before this section is expanded further — the naming similarity ("attestation") is suggestive but not itself confirmation of a functional link.

# Chapter 8: Worker Instances and the Spine/Worker Architecture

*Source: chandra-worker-diagnostics.html; chandra-instance-dashboard.html (new tool, v31c30); chdev-reload.lisp, and chandra/instances/{9101,9102,9103,9104,9195}/ (start-enterprise.lisp, run-ent.sh) from the source zip.*

A Chandra deployment is physically realized as one or more **worker instances** — independent OS processes, each with its own AllegroCache database, each serving one physical isolation surface (Chapter 5) that the Spine Editor (Chapter 13) packed one or more logical Levels (Chapter 4) into.

## One process, one database, one instance directory

Confirmed directly from the source zip rather than inferred from the UI: each worker instance lives in its own directory under chandra/instances/<port>/ — five instances were present in the reviewed snapshot, at ports **9101, 9102, 9103, 9104, and 9195**. Each instance directory carries its own start-enterprise.lisp, run-ent.sh, and its own chandra-db-<port>/ AllegroCache database directory. chdev-reload.lisp confirms the database path is configuration-driven, not hardcoded: *chandra-db-dir* defaults to "chandra-db/" but is overridable per-process via chandra.conf's :db-dir key, and start-enterprise.lisp resolves its own source root and instance root independently (CHANDRA_SOURCE_ROOT / CHANDRA_INSTANCE_ROOT environment variables, falling back to the process's own load-path and working directory) — meaning shared source code can be deployed once while each instance still gets its own mutable, isolated instance state. This is the literal mechanism behind Chapter 5's "T cancels out of S*" result: scaling out is adding more of these independent instance directories at the same size, not growing one instance's database.

## What NOT to do to a running instance's database — a documented incident

start-enterprise.lisp's own comments preserve a real incident worth restating here, because it's a sharp, specific operational warning rather than generic caution: an earlier version of the boot sequence closed and reopened the AllegroCache database on every boot, on the theory that AllegroCache might have its own thread-level staleness independent of AllegroServe's worker pool. That theory may have been right, but the fix was not safe — after a close/reopen cycle, HTTP requests to certain routes began failing with an AllegroCache internal error (BACKEND-LAST-TRANS with args (NIL)), even though direct REPL reads confirmed the underlying data was intact. The corruption was written to the database's **on-disk structures** during the close/reopen, persisted across a full process restart, and required deleting and recreating the database to recover. The current boot sequence's explicit comment: *"Do not reintroduce AllegroCache close/reopen here without real documentation or vendor guidance on this."* If you are ever debugging instance staleness, this specific fix is not the answer, and the comment exists specifically so nobody has to relearn this the hard way.

## Instance identity: /api/enterprise/self

Every worker instance exposes its own identity manifest at /api/enterprise/self (documented fully in Chapter 9) — instance name, immutable ID, environment, build badge, and (as of a recent addition) personality_id, the clean identifier of whichever personality (Chapter 6) seeded that instance's spine (e.g., "cmmc-defense-7level" rather than a raw filename). This one endpoint is the shared foundation for three separate tools:

**Worker Diagnostics** (chandra-worker-diagnostics.html) — local, non-polling diagnostic surface; explicitly does not depend on CEE or the spine instance being reachable, so it works even when the instance it's diagnosing is unhealthy.
**Instance Dashboard** (chandra-instance-dashboard.html, new as of this session, build v31c30) — a per-worker dashboard whose title literally renders as INSTANCE DASHBOARD — <instance_name>, with the name pulled live from the same self manifest. The browser tab title becomes CHANDRA · Instance Dashboard — <INSTANCE_NAME> — meaning if you have several worker dashboards open in different tabs, each tab is self-identifying without needing to check the URL.
**Marshaller** (Chapter 7) — reads the exact same manifest remotely, at VERIFY time, to populate its fleet directory.

## Why workers are the physical realization of isolation surfaces

Chapter 5 establishes the math ($S^*$ is population-independent; scale by adding boundaries, not by growing one). Chapter 6 establishes that the Industry Configurator never touches physical deployment at all. Chapter 13 establishes that the Spine Editor is the tool that actually computes instance count and packs Levels into instances. This chapter is where that math becomes concrete: a "worker instance" *is* one of Chapter 5's isolation boundaries, realized as an actual OS process with its own database, its own acache (Chapter 18), and its own identity manifest. When this manual talks about an isolation surface being "sized," it means, physically, deciding how many of these worker-instance directories to stand up and which logical Levels each one hosts.

## Known incomplete feature: Andon

The Spine UI (visible live on vatican.chandrahub.net) includes an "ANDON ALL" control alongside its dashboard. Per James directly: **this functionality is not fully developed yet.** Don't document Andon as a working operational feature in this or any other chapter until that's confirmed otherwise — it should stay a named-but-unelaborated control until there's real behavior behind it to describe.

# Chapter 9: Integration and Migration — Working With the Chandra API Directly

*Reference build: chandra-enterprise-editor.html, badge v31c425 (2026-08-09). Every endpoint, payload shape, and response field in this chapter is taken directly from CEE's own client code — this is the same API CEE itself calls, not a separate integration surface. If you're building a migration script, a sync job, or a wrapper around existing infrastructure, you are using exactly what the browser does.*
*Note: James — you mentioned you've written integration notes elsewhere already. This chapter is built strictly from what's provably in the shipped client code, so treat it as the "ground truth from the wire protocol" half; fold in your existing notes wherever they add context (auth/deployment topology, worker routing specifics, etc.) that isn't visible from the client alone.*

---

## 9.1 What "the API" actually is

There is no separate public API distinct from what CEE calls. Every action in the browse toolbar (Chapters 1–10) is a plain HTTP call to a JSON REST endpoint under /api/enterprise/* (plus a small /api/health and /api/enterprise/self surface for instance identity). CEE is a thick client over this API — nothing happens server-side that CEE's JavaScript couldn't equally call from a script, a migration tool, or another application entirely.
That means integrating Chandra into existing infrastructure is, at the protocol level, just calling these same endpoints from wherever you need to — a cron job, an ETL pipeline, a webhook handler, another product's backend.

**Base URL.** Every endpoint is relative to an *instance address* — the worker (or spine) you're talking to. CEE resolves this per-Hub via levelBase(lv); there is no single fixed base URL system-wide, because a given deployment may have multiple worker instances (Ch. 8) each fronting a different isolation surface. When you write an integration, you supply the instance address explicitly — it's the same value CEE shows as "instance address" in diagnostics.

**Auth, as observed from the client.** CEE's own request helpers (getJson/postJson) send only Content-Type: application/json and Accept: application/json — no bearer token or API key header is added client-side. Every mutating call does carry an operator field in its JSON body (see §9.3), which is how a human/service identity gets attributed to the resulting CU, but that's an attribution field, not an authentication credential. **This chapter documents the request/response contract, not the deployment's access-control layer** — how a given instance is actually gated (network placement, reverse-proxy auth, mTLS, etc.) is an infrastructure decision outside what's visible in this client file. If you've already written notes on that elsewhere, that's the piece to bring in here.

**Every response is JSON**, and CEE's own parsing convention is worth adopting in your own integration: a non-2xx HTTP status *or* a {"ok": false, …} body (even on HTTP 200) both mean failure, and the message/error field carries the human-readable reason.

---

### 9.2 Reading data: listing and fetching records

**List records in a Hub:**
GET {base}/api/enterprise/records/list?hub_id={hub_id}&status=active
GET {base}/api/enterprise/records/list?hub_id={hub_id}&parent_subject_id={parent_id}&status=active
hub_id — required. The governed table you're reading from (CEE calls this hubIdForLevel internally — a Hub, not the Spine tree-node id).
parent_subject_id — omit entirely for top-level/root records. This is important: the backend (crud-list-spoke-records) treats a *missing* parent_subject_id as "give me records with no parent scope at all" — sending an empty string or a placeholder is not the same thing and will not behave as "no filter."
status — lifecycle filter (active is what the browse defaults to; other lifecycle states exist per Ch. 1's soft-delete/archive model).

```
async function listRecords(base, hubId, parentSubjectId) {
 const q = new URLSearchParams({ hub_id: hubId, status: "active" });
 if (parentSubjectId) q.set("parent_subject_id", parentSubjectId);
 const r = await fetch(`${base}/api/enterprise/records/list?${q}`, {
  cache: "no-store",
  headers: { Accept: "application/json" }
 });
 const j = await r.json();
 if (!r.ok || j.ok === false) throw new Error(j.message || j.error || `HTTP ${r.status}`);
 return j;
}
```

**Fetch instance identity** (useful as a first call in any integration, to confirm you're pointed at the instance you think you are, and to get its immutable instance ID for logging):
GET {base}/api/enterprise/self
**Health check** (confirms the worker is reachable and its backing database is open — CEE polls this for every worker tier's status pill):
GET {base}/api/health
A healthy response is HTTP 200 with {"ok": true, "instance_id": "…", "instance_name": "…", "db_open": true, …}. db_open: false means the process is up but its database isn't — CEE surfaces this distinctly as "DB CLOSED (restart required)" rather than treating it the same as unreachable, and your integration should probably do the same.

---

### 9.3 The operator field and the optimistic-lock pattern

Two conventions show up on **every** mutating call in this API, and both matter for a correct integration:
**operator** — a plain string identifying who's making the change, sent on every create/update/delete call. In the CEE client it defaults to a value read from localStorage.getItem("chandra.operator"), falling back to "cee-acceptance-user" if unset. For a script or service integration, set this to something meaningful and stable (e.g. "migration-script-v1", "salesforce-sync") — it becomes part of the permanent CU attestation for anything that endpoint creates or changes, so it's your integration's identity in the audit trail going forward. Don't reuse a human operator string for automated writes; give the integration its own.

**loaded_tail_hash** — Chandra's optimistic-concurrency-control token. Every record carries a hash-chain tail; to update or delete a record you must supply the tail hash *you last read*, not a placeholder. The

server rejects the write if that tail no longer matches the record's current state (i.e., someone else changed it since you read it), which is the mechanism that makes concurrent writes from multiple integrations safe rather than silently clobbering each other. This is also why a read (records/list or a record's own current state) has to precede any write in an integration — you cannot construct a valid update without first fetching the record's current tail.

---

### 9.4 Creating records

```
POST {base}/api/enterprise/records/create
Content-Type: application/json
{
  "hub_id": "...",
  "parent_subject_id": null,
  "level_number": 2,
  "target_level_number": 2,
  "name": "New Record",
  "values": { "field_key_1": "...", "field_key_2": 123 },
  "operator": "migration-script-v1"
}
```

parent_subject_id — null for a root-level record; the parent's subject_id to attach under a parent (this is how related-table/portal parent-child linkage from Ch. 7 is actually established).

values — an object keyed by the field's indexed name, matching the Hub's current published form design. Values that don't validate against the form design (wrong type, a required field missing) are rejected server-side, the same validation path CSV import (Ch. 5) and interactive save go through — there's no separate lenient path for API callers.

The response's record (or the top-level object, depending on endpoint — see the unwrapping pattern CEE itself uses below) carries the new subject_id and its initial tail hash, both of which you need if you're about to make a follow-up call (e.g. uploading a file attachment, §9.6).

```
async function createRecord(base, payload) {
  const r = await fetch(`${base}/api/enterprise/records/create`, {
    method: "POST",
    cache: "no-store",
    headers: { "Content-Type": "application/json", Accept: "application/json" },
    body: JSON.stringify(payload)
  });
  const j = await r.json();
  if (!r.ok || j.ok === false) throw new Error(j.message || j.error || `HTTP ${r.status}`);
  // CEE's own unwrapping pattern — the created record may be nested under `.record`
  const record = (j && j.record) || j || {};
  const subjectId = String(record.subject_id || record.record_id || record.id || j.subject_id || "");
  return { response: j, record, subjectId };
}
```

---

### 9.5 Updating and deleting records

**Update** — requires the tail hash from your most recent read of that record:

POST {base}/api/enterprise/records/update

```
{
  "subject_id": "...",
  "loaded_tail_hash": "...",
  "values": { "field_key_1": "new value" },
  "operator": "migration-script-v1"
}
```

**Soft-delete** (the standard delete path — recoverable, per the lifecycle/archive model in Ch. 1–2):

POST {base}/api/enterprise/records/delete

```
{
  "subject_id": "...",
  "loaded_tail_hash": "...",
  "operator": "migration-script-v1"
}
```

**Archive / restore / restore-to-deleted** exist as separate endpoints (/api/enterprise/records/archive, /api/enterprise/records/restore, /api/enterprise/records/restore-to-deleted) for moving records through the fuller lifecycle states beyond a simple soft-delete — same subject_id + loaded_tail_hash + operator shape.

A missing or stale loaded_tail_hash is the single most common integration failure mode worth designing around defensively: always read-immediately-before-write in a migration script rather than caching a tail hash from an earlier stage of a long-running job, since the record may have changed underneath you (by CEE, by another integration, or by your own prior step) between the read and the write.

---

### 9.6 File attachments

Attachments are their own small endpoint family, bound to a record's subject_id rather than embedded in the record payload itself — the field's *value* just stores the filename/metadata; the bytes live behind these endpoints:

POST {base}/api/enterprise/attachments/upload

```
{
  "subject_id": "...",
  "filename": "invoice.pdf",
  "content_base64": "...",
  "content_hash": "sha256-hex-of-file-bytes",
  "operator": "migration-script-v1"
}
```

10MB hard limit per file in CEE's own client (worth matching in an integration, though the authoritative limit is whatever the backend enforces). content_hash is a SHA-256 of the raw file bytes, computed client-side before upload — CEE computes and sends it for integrity verification but doesn't hard-fail locally if hashing fails, so treat it as strongly recommended rather than strictly required.

POST {base}/api/enterprise/attachments/delete

```
{ "subject_id": "...", "filename": "invoice.pdf", "operator": "migration-script-v1" }
```

GET {base}/api/enterprise/attachments/download?subject_id={id}&filename={name}

Note the two-step pattern CEE itself uses when a new record has a file field: **create the record first, upload the file second, then a follow-up records/update writes the real filename into the field value**

(using the tail hash returned by the create call). A file field's value is provisional until that second update lands — don't treat a bare records/create response as having a fully-resolved file field.

### 9.7 Reading provenance: the CU chain

Every governed mutation produces a CU (Chandra Unit — Ch. 1's append-only attestation), and the chain is independently queryable, which is the piece that makes this genuinely useful for migration verification (confirming a batch import actually attested, not just that the HTTP call returned 200):
GET {base}/api/enterprise/chain/index?subject_id={subject_id}
Returns the list of CUs for a given record/subject — its full history.
GET {base}/api/enterprise/chain/detail?cu_id={cu_id}
Returns one CU's full detail — the attested artifact itself.
A migration script that wants a real audit trail (not just "the API said OK") should, at minimum, capture the cu_id/tail hash returned from each write and spot-check a sample against chain/detail — this is the same verification path CEE's own historical-CU viewer uses.

### 9.8 Bulk ID pre-allocation

If a migration needs to mint canonical record IDs ahead of the actual write (for example, to build cross-references between records before any of them exist yet):
GET {base}/api/enterprise/ids/new?count={n}
Returns {"instance_id": "...", "ids": [...]}. IDs are instance-scoped (prefixed/derived from the issuing instance's own identity), so pre-allocated IDs from one worker instance are not portable to another — request them from the instance you'll actually be writing to.

### 9.9 A minimal end-to-end migration example

Putting the pieces together — importing external records into a Hub, with verification:

```
const BASE = "https://your-worker-instance.example.com";
const OPERATOR = "migration-script-v1";

async function migrateRecord(hubId, externalRow) {
 // 1. Create
 const { record, subjectId } = await createRecord(BASE, {
  hub_id: hubId,
  parent_subject_id: null,
  level_number: 2,
  target_level_number: 2,
  name: externalRow.name,
  values: {
   external_id: externalRow.id,     // preserve source-system provenance as a plain field —
                                    // never as the canonical subject_id; Chandra always
                                    // mints its own identity on create (same rule CSV
                                    // import in Ch. 5 follows)
   status_field: externalRow.status,
  },
```

```
  operator: OPERATOR
 });

 // 2. Verify the write actually attested
 const chain = await fetch(`${BASE}/api/enterprise/chain/index?subject_id=${subjectId}`,
  { headers: { Accept: "application/json" } }).then(r => r.json());
 if (!chain || !chain.length) throw new Error(`No CU attested for ${subjectId}`);

 return { subjectId, cuId: chain[chain.length - 1].cu_id };
}
```

This mirrors, at small scale, exactly what CEE's own CSV import batch loop does internally: create through the ordinary record endpoint, never write the source system's ID into the canonical identity field, and treat "the write returned 200" and "a CU actually attested" as two different things worth checking separately.

---

| | | |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

| | | |
|---|---|---|
| | | |
| | | **9.10 Endpoint reference (as verified in this build)** |

**Endpoint**
**Method**
**Purpose**
/api/health
GET
Liveness + DB-open check for a worker instance
/api/enterprise/self
GET
Instance identity manifest (name, immutable ID, environment, spine state)
/api/enterprise/records/list
GET
List records for a Hub (hub_id, optional parent_subject_id, status)
/api/enterprise/records/create
POST
Create a record
/api/enterprise/records/update
POST
Update a record (requires loaded_tail_hash)
/api/enterprise/records/delete
POST
Soft-delete a record
/api/enterprise/records/archive
POST
Move a record into the archive lifecycle state
/api/enterprise/records/restore
POST
Restore an archived/deleted record
/api/enterprise/records/restore-to-deleted
POST
Step a record back to the deleted (not archived) state
/api/enterprise/records/deleted
GET
List soft-deleted records
/api/enterprise/records/archived
GET
List archived records
/api/enterprise/attachments/upload
POST
Upload a file bound to a record’s subject_id
/api/enterprise/attachments/delete
POST
Delete a bound file

/api/enterprise/attachments/download
GET
Download a bound file
/api/enterprise/chain/index
GET
List CUs for a subject (its full history)
/api/enterprise/chain/detail
GET
Fetch one CU's full attested detail
/api/enterprise/ids/new
GET
Pre-allocate canonical IDs from an instance
/api/enterprise/hub/form-design/latest
GET
Fetch a Hub's currently published form design
/api/enterprise/hub/form-design/save
POST
Publish a form design change
/api/enterprise/sub-hub/create
POST
Create a governed sub-Hub (related table, Ch. 7)
/api/enterprise/spine/environment/status
GET
Spine/environment status for a given environment label

This list is everything directly observed in the current client build — it is not necessarily the complete server-side API surface (Test Plans, saved searches, and help-context endpoints exist too but are CEE-internal tooling rather than data-integration surfaces, so they're omitted here as out of scope for a migration chapter).

---

*This chapter covers the wire protocol only. Deployment topology — which instance fronts which isolation surface, how routing/auth is actually enforced at the network layer, and how this maps onto the worker architecture in Ch. 8 — is the next layer up, and is where your existing notes should slot in.*

**A sharper public framing of this same integration story, confirmed live from chandrahub.net (Ch. 10):** *"Integrating an existing application with Chandra requires one column addition per auditable database table. No schema redesign. No audit middleware. No separate compliance system. Every write your application makes appends a context unit to the corresponding Chandra chain. The chain is the compliance record."* This is a more citable, examiner-facing version of the same claim this chapter makes operationally — worth using directly when explaining the integration story to a non-technical stakeholder rather than re-deriving it from the endpoint reference above.

---

# Chapter 10: Chandrahub — The Governed Deployment Network

*Source: chandrahub.net homepage (live-fetched, 2026-08-23); marshaller.chandrahub.net and vatican.chandrahub.net (live-fetched, confirming the multi-org pattern); James, directly, on the narrative positioning below. The doc subpages linked from chandrahub.net's own nav (Understanding Chandra, Being Audited, Auditing with Chandra, Examiner Reference, Standards, Compliance) all returned HTTP 404 as of this fetch — only the homepage is live. Revisit this chapter once those pages exist.*

Chandrahub is described on its own homepage as **"the forest authority for every governed Chandra deployment."** It sits above the Marshaller (Chapter 7) conceptually — Marshaller tracks one organization's (or GR's own) fleet of spine addresses; chandrahub is the broader, public reference point that any Chandra deployment, anywhere, can point an examiner at.

## What chandrahub.net actually is — and, importantly, what it is not

chandrahub.net itself is **the open beta** — General Reasoning's own public instance, the entry point for organizations evaluating or beginning a Chandra deployment. Confirmed live and reachable: vatican.chandrahub.net, lockheed.chandrahub.net, and anthropic.chandrahub.net are each running what appears to be a full spine (the Vatican instance shows the same Acceptance/Canary/Production toolbar documented in Chapter 2, at build v31c444).

**This is where the narrative needs to be stated carefully, per James directly, rather than left to the subdomain examples to imply on their own:** chandrahub.net is simply the site hosting the public beta test. It is technically *capable* of multi-tenancy — the subdomain pattern above could look like a multi-tenant SaaS offering at a glance — but that is explicitly **not** the story this manual tells, and not the end-state General Reasoning is selling. Shared tenancy is itself a violation of the CRC standard's own Consolidate/Close pillars (Chapter 5): a shared hosting boundary is exactly the kind of unowned, traversable edge CRC exists to eliminate, regardless of how well-isolated the tenants' *data* is within it (see the next section). Standalone servers — for example, dedicated instances on Linode — are a legitimate interim deployment shape for a client not yet ready to run their own hardware. But the actual end-state offer to a client that wants real control and security is a **sovereign, self-hosted deployment running their own AegisGenera boundary** (Chapter 19) — not a seat on General Reasoning's shared infrastructure, however well-partitioned. Document chandrahub.net's multi-org subdomains as a beta convenience and a demonstration of the architecture's generality (Chapter 6 makes the same point about the Vatican personality), not as evidence of a multi-tenant product direction.

## Data isolation as both a security posture and a disaster-recovery strategy

Every organization's data on chandrahub — and in the Chandra architecture generally — is isolated, which directly serves the CRC/ISS formula (Chapter 5) and enables fine-grained segmentation. It's worth naming the second reason this matters, stated by James directly, because it's a distinct argument from the security one and easy to under-weight: **isolating each organization's chronicle (its CU chain history, Chapter 1) independently gives the best realistic chance of recovering it intact, regardless of that organization's complexity, physical size, or database size.** A monolithic shared store makes disaster recovery an all-or-nothing proposition — one corruption event, one bad restore, one operational mistake can threaten every tenant's history at once. Per-organization isolation means a DR event's blast radius is bounded to the organization(s) actually affected. This is the same underlying posture — isolate everything down to fine-grained, independently-recoverable segments — that also drives Chapter 18's

Help-content architecture; security and disaster recovery turn out to want the same shape of system for related but distinct reasons.

## The examiner reference: stable regardless of where a deployment runs

Chandrahub.net's stated role, regardless of whether a given deployment is the open beta or a client's own sovereign instance: **it is the stable public reference the examiner reference lives at permanently.** It's the URL printed on a GABA cert (Chapter 11); it's where an examiner starts, and it does not move even if the underlying deployment they're examining is running entirely on the client's own infrastructure elsewhere.
The site names three distinct audience-specific documentation tracks, planned but not yet live as of this fetch: - **Examiner Reference** — for regulators verifying GABA certs, reading CUs, and tracing audit opinions to work papers, written for someone with no prior Chandra knowledge. - **Auditing with Chandra** — for auditors, covering SSAE 18 engagement types, governed work papers, opinion issuance, and chain of custody. - **Being Audited** — for organizations, covering what their audit trail contains, how to produce evidence for examiners, and retention policies by standard.

## Terms confirmed publicly for the first time here

Two pieces of vocabulary get their first plain-language public statement on this site, worth using consistently rather than the internal shorthand elsewhere in this manual:
**CU** is spelled out as **"context unit."** Internal code and most of this manual use "CU" without expansion; chandrahub.net is the first source that states the full term for a non-technical reader.
**chandrapassport** — referenced in the site's own footer integrity note: *"Page integrity: Pending chandrapassport deployment — CU verification active post-launch."* This is the first evidence that Chandra Passport (tracked elsewhere as an internal RBAC/bonding architecture) has a **public-facing role**: verifying the integrity of chandrahub.net's own published content via CU, once deployed — meaning the examiner reference itself is intended to eventually be a CU-attested artifact, not just a static page describing CU-attested artifacts elsewhere.

## Known incomplete feature

The Spine UI's **"ANDON ALL"** control, visible live on vatican.chandrahub.net, is not fully developed yet (confirmed by James directly — see also Chapter 8). Don't describe it as operational in this chapter or elsewhere until that changes.

## Open items for this chapter

The doc subpages (Standards, Compliance, and the three audience tracks above) should be re-fetched and folded in once they're live — this chapter currently reflects only what the homepage states, plus direct clarification from James on positioning. The relationship between chandrahub's attestation model and the Marshaller Attestation Console (Chapter 7, Chapter 16) also has not been confirmed and shouldn't be assumed from naming alone.

# Chapter 11: The GABA Standard

*Source: crcstandard.org (nav bar link to gabastandard.com; Pillar 04 "Boundary" scoring criteria, live-fetched). gabastandard.com itself has not yet been fetched or reviewed — this chapter is deliberately thin rather than speculative, and should be substantially rewritten once that site is reviewed directly.*
GABA is referenced as its own standard, at its own domain (gabastandard.com), and cross-linked from CRC's own site navigation — meaning it's a sibling standard to CRC, not a CRC sub-component, even though the two are clearly designed to interlock.

## What's confirmed so far

GABA's role is named precisely at one specific point in the CRC framework: **Pillar 04 (Boundary)'s top scoring tier, "Attested."** The distinction between "Hardened" (tier 3 — cert-pinned endpoints, request signing, anomaly detection, circuit breakers) and "Attested" (tier 4, the maximum) is specifically that **GABA attestation is complete for every endpoint**, residual risks are formally documented and signed by an authorized signatory, and — the detail that most sharply defines GABA's scope — **computer-use AI agents require explicit GABA attestation separate from inference-only endpoints.**
That last point is the clearest signal available about what GABA actually governs: it is not a general AI-safety attestation scheme, but one specifically concerned with the distinction between an AI system that merely returns text (inference-only) and one that can take actions in a computer-use capacity. An inference endpoint and a computer-use agent hitting the same governed boundary are treated as categorically different risks under this framework, and GABA attestation is the mechanism for the latter, more consequential category.

GABA certs are also the artifact referenced on chandrahub.net (Chapter 10) as printing the chandrahub.net URL for examiner verification — meaning GABA attestation and chandrahub's examiner-facing role are meant to work together, though the precise mechanics of that handoff (what a GABA cert actually contains, how an examiner verifies one against a CU chain) have not been sourced directly yet.

## What this chapter does not yet cover

GABA's full scoring criteria or certification process (unlike CRC's 0–16 scale, no equivalent structure has been reviewed for GABA)
Whether GABA is MIT-licensed like CRC and Chandra Protocol, or has different terms
The acronym's expansion — "GABA" has not been spelled out in anything reviewed so far
AegisGenera's likely role as GABA's technical enforcement layer (Chapter 19 flags this as a reasonable inference from CRC's own Pillar 04 description, but it is explicitly unconfirmed against AegisGenera's own source, which doesn't exist yet)
This chapter should be rewritten from gabastandard.com directly before it's treated as more than a placeholder.

# Chapter 12: General Database Functionality (Business Record Browses)

*Reference build: chandra-enterprise-editor.html, badge v31c425 (2026-08-09). This chapter documents the Business Records browse — the FileMaker-style window you get from opening a Hub's records — and the ten core capabilities below. Everything described here was verified directly against the shipped UI and JS in that file, not assumed from the naming scheme.*

Every business-record browse in CEE shares one toolbar. Left to right: **Add / View / Edit / Duplicate / Soft-delete** as always-visible icon buttons, then five menu launchers — **Find**, **Found Set**, **Sort/View**, **Hub**, **Form**, **Trigger**, **Test**, **Import/Export** — followed by **Refresh** and a live search box. All ten capabilities in this chapter live inside that one toolbar; nothing in this chapter requires leaving the browse window.

A running status line under the toolbar always reads: <found count> · <sort summary> · Query plan: local scan · <n> selected. Treat that line as a state readout — it is the fastest way to tell whether you are looking at the full table or a found set, and in what order.

---

### 1. Sort

**Where:** Sort/View menu → **Sort Records**, or click any sortable column header directly.
Sort in CEE is a draft-based operation with true Apply/Cancel isolation — nothing changes on screen until you explicitly click **Sort** in the dialog. Opening the dialog shows one row per active sort key; **+ Add Sort Field** appends another. Multiple keys are evaluated top to bottom (a genuine multi-key sort, not a single field with tiebreakers bolted on), and each row can be reordered with the ↑ / ↓ controls or removed with ×.

| | |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |
| | Per sort key you control: |

**Control**
**Effect**
Field
Any sortable field on the current form design (text, number, date, boolean, select types)
Direction
Ascending / Descending
Empty placement
Empty values sort first or last, independent of direction
Natural

Natural (numeric-aware) string comparison vs. strict lexical
Case sensitive
On/off — off by default
Select ordering
For Select fields only: sort by current label text, or by the field's defined option order
Quick column-header sorting is layered on top of the same mechanism: clicking a sortable header sets it as the sole sort key; shift-clicking (additive) extends the existing sort with that column as an additional key; clicking an already-sorted header toggles its direction. There's no separate "quick sort" data path — it's the same sortSpec the dialog edits, so the two are always consistent with each other.

Sorting operates on the **current found set**, not the whole table — if you've constrained down to 12 records, Sort only reorders those 12. The active sort is shown in the Sort/View button's own label (Sort: <field> ↑) and in the status line, and it persists across Refresh and lifecycle-view changes until you change it or clear it.

If a sort key references a field that no longer resolves (retired, renamed, or type-changed since the sort was set), CEE flags it as **Missing field** in the dialog rather than silently dropping or misapplying it — you have to resolve or remove it before sorting again. Any other resolution problems (an unparseable date or non-numeric value in a numeric-typed field, for instance) surface as inline diagnostics rather than corrupting the row order.

---

### 2. Found-Set Operations

**Where:** the **Find** menu (search/constrain/extend) and the **Found Set** menu (omit/omitted-only/return/show all) — two menus working over one shared found-set state.
CEE's found-set model is the direct FileMaker analogue: a business browse always has a *base* set (everything, or the result of your last Find) and an *omitted* set layered on top of it. The two menus split responsibility cleanly:

**Find menu:** - **Find Records** — opens the Find dialog fresh, or reopens your last active criteria if a found set already exists. - **Saved Searches** — CU-backed saved search definitions you can name, re-run, and manage independently of the current browse state. - **Constrain Found Set** — runs a new Find *inside* the current found set (intersection). Disabled until a found set is active. - **Extend Found Set** — runs a new Find and unions the result into the current found set, with automatic de-duplication. Also requires an active found set first. - **Show All Records** — the hard reset: clears findActive, the base ID set, and all omissions in one step.

**Found Set menu:** - **Omit Selected** — removes the currently selected row(s) from view without touching the underlying records. If no found set is active yet, Omit Selected implicitly promotes "everything" into the base set first, so omission always has something concrete to subtract from. - **Show Omitted Only** — flips the browse to show *just* the omitted records, for review. - **Return to Found Set** — flips back to the included view. - **Show All Records** — same hard reset as in the Find menu, exposed here too for convenience.

Buttons are gated to valid states automatically: Omit Selected is disabled with nothing selected or while already viewing the omitted set; Constrain/Extend are disabled with no active find; Show Omitted Only is disabled with nothing omitted. You don't have to remember the legal sequence — the toolbar enforces it.

**A word on reliability here:** the found-set fields (foundBaseIds, omittedIds, foundView, findOperation) are the single source of truth that every later Refresh recomputes against. A past bug (fixed) briefly let a post-import view get into a state where the toolbar *looked* like a found set but the underlying fields weren't set consistently, so Refresh kept re-deriving an empty result from stale state. That specific failure mode is closed now, but the underlying lesson is durable and worth knowing: if a browse ever looks "stuck" showing nothing when you know records exist, **Find menu → Show All Records** is the unconditional reset, and it does not require a reload.

---

### 3. Record Navigation

**Where:** navigation controls on the business browse, operating over whatever is currently on screen.
First / Previous / Next / Last step through the **current visible found set, in the current sort order** — not the whole table. Each navigation action selects exactly one canonical record (never a range) and the browse shows your position as **Record n of m**. Buttons disable themselves at the boundaries (Previous/First disabled on record 1, Next/Last disabled on the last record), and the newly-selected row is scrolled into view automatically so you never lose visual track of where you are, especially in a long found set.

Because navigation reads the same found-set/sort state as §1 and §2, the three chapters compose the way you'd expect from FileMaker: sort first, constrain the found set second, and First/Next/Last will walk exactly that filtered, ordered sequence — no separate configuration needed.

---

### 4. Duplicate Record

**Where:** the **Duplicate** icon button in the always-visible toolbar row.
Duplicate is enabled only when exactly one **active** record is selected and the Hub has a published form design (the same form-design gate that governs Add and Edit). Clicking it opens the standard record editor pre-loaded with a deep-copied snapshot of the source record's field values — not a reference to the source, a genuine value copy, so editing the draft can never mutate the original.

What is deliberately **not** copied: - Canonical record identity (the duplicate gets a fresh ID on creation, through the normal record-genesis path — same as a brand-new record) - CU history / chain — the duplicate starts its own provenance chain from its own genesis - Parent linkage is preserved (the duplicate stays attached to the same parent record), but it is not a reference back to the source record

The draft retains source-record and tail provenance internally for audit purposes, but that's metadata about *how the draft was created*, not a live link. Cancel is fully inert — if you cancel out of the duplicate editor, nothing is written anywhere and the source record is untouched.

On save, the new record goes through the ordinary create path, the browse reapplies the current found set and sort, and the new record is selected automatically **if it falls within the current found set**. If it doesn't — say you're duplicating from within a constrained view and the copy's field values would put it outside that constraint — CEE tells you explicitly ("Duplicate created, but it is outside the current found set") rather than leaving you to wonder where it went.

---

### 5. Import / Export

**Where:** the **Import/Export** menu → **Export / Report (CSV · PDF)…** / **Import CSV…**.
**Export** opens the **Export / Report** dialog, which as of this build handles both CSV and PDF from one place — see §9 for the PDF half (layout modes, generation, and attestation). For CSV specifically: it exports whichever scope you choose (found set / tagged / current record — §9 has the full scope picker breakdown) in the current sort order, with display-formatted values (select fields export their current label text, not raw option IDs), a SHA-256 digest of the output, and a CU-attested export job so it's part of the governed audit trail rather than a side-channel file dump.

**Import** is **add-only** in the current build — there is no update/upsert path yet. Every accepted row is created through the ordinary CU-backed record-create endpoint; nothing is ever an in-place overwrite of an existing record. The flow is:

**Import CSV** → choose a file → **Read CSV**. Blocked until the Hub has a published form design.
**Map CSV Columns** — CEE auto-matches header names to form fields by label or field key, and always treats identity-looking columns (record_id, canonical_record_id, source_record_id) as source provenance only — they're never imported as data and never used to overwrite an existing record. Every accepted row gets a fresh canonical record ID and its own genesis CU, unconditionally.
Choose a **commit policy**: *Commit valid rows only* (skip and report failures, keep the good rows) or *Stop on first write failure*.
**Validate & Preview** — every row is checked against the form design's real validation rules before anything is written: required fields, type coercion, and select-field resolution (a select cell that matches a current option *label* is resolved to that option's ID; text that already looks like a raw ID passes through unchanged; nothing is silently rejected just because it doesn't match a current label — see the label/ID handling below).
**Commit** — writes the valid rows, then shows **Import Completed**: created count, rejected count, a per-row error table for anything rejected, the job ID, and the CU attestation (or an explicit attestation-failure notice if the job couldn't be attested). The browse automatically switches to show the affected-record found set afterward.
A few import behaviors worth knowing because they were hard-won: - **Required fields:** if the form design's name field is itself required (common for canonical_label-style base fields), the imported value for that field is preserved in the row's values — it is not silently stripped before the create call. - **Select fields:** CSV text is resolved against current option labels where a match exists (this also transparently "heals" an option that's since been renamed but is still present); otherwise the raw text is passed through as-is on the assumption that it's already a valid option ID. Nothing is force-cleared or rejected purely for not matching a current label. - **Duplicate-file guard:** if the same source file's digest appears to have already been imported into this Hub in this browser before, you get a confirmation prompt rather than a silent double-import.

---

## 6. Validation and Auto-Entry

This is the one chapter heading in James's list where it's worth being precise about *where* the capability actually lives, because it's split across two parts of the product rather than living in one obvious "auto-entry" panel.

**Field-level validation, in the Form Designer (_form_design):** every field carries Required and Indexed toggles you set when designing the field, plus a Case control (None / UPPERCASE / lowercase / Title Case) on text fields specifically. Case normalization applies live as you type in the record editor — the input field re-cases its own content on every keystroke via applyTextCase, so what you see while entering data is what gets saved, not a silent server-side rewrite you'd have to guess at. Required-field enforcement is checked both at interactive save time and, separately, at CSV import time — a required field with no value blocks the row in both paths, with an explicit "is required" message rather than a generic failure.

**Auto-entry and cross-field validation logic, via governed Trigger Definitions:** CEE does not currently expose a dedicated "auto-entry type" dropdown inside the Form Designer's field editor. Instead, auto-entry behavior (stamping a creation timestamp, the creating principal, a literal default, a UUID, or a monotonic sequence number) and validation logic beyond Required/Indexed (a conditionally-required field, an outright rejection rule) are configured as **governed executor actions on a Trigger Definition**, attached to an event like on-create or on-validate. See §10 below for the full editor — the two allowlisted validation executors (validation.require@v1, validation.reject@v1) and the field-stamping executors (field.set-timestamp@v1, field.set-principal@v1, field.set-uuid@v1, field.set-sequence@v1, field.set-literal@v1, plus the two text-normalization executors) are what implement this chapter's auto-entry half in practice.

**Current gap, stated plainly:** there is a real, tested acceptance-fixture harness (cee-validation-autoentry-fixture-runner) that exercises auto-entry semantics — normalization, defaulting, generation, matching the current principal, monotonic sequencing, and correct request rejection with no record created — but that harness drives the *Trigger Definition* executors described above, not a separate standalone "auto-entry" surface. If you're looking for a single dedicated auto-entry panel distinct from Trigger Definitions, it doesn't exist yet as of this build; the governed-executor Trigger Definition editor in §10 is the actual mechanism.

---

## 7. Related-Record Portals

**Where:** the **Related Tables** pane, permanently docked on the left side of every business browse, with a draggable splitter to resize it.

This is CEE's equivalent of a FileMaker portal, implemented at the table level rather than as an embedded row-list: select a parent record in the main grid, and Related Tables lists every child Hub attached under it, each showing status, last-updated, and a live child-record count. Clicking a related table opens a *filtered* browse scoped to that parent record — the same full-featured business browse described everywhere else in this chapter (its own sort, its own found set, its own toolbar), just pre-constrained to the selected parent.

Related Tables carries its own governance menu (**+ Create Table**, and a **More ▾** menu with Hub Info, Open Form Designer, Duplicate Table Definition, Remove Relationship, Retire Hub, and Request Purge), so table-definition work and record-browsing work stay in the same pane without forcing a context switch to the Spine Editor for routine related-table administration.

The splitter width is remembered per browse window; double-clicking it resets to the default split.

---

### 8. Calculated Fields

**Where:** Form Designer (_form_design) → field editor → **Calculated** toggle, on any Number (integer) field.
*Correction from an earlier draft of this chapter: this capability was verified against build v31c390 and found absent. It is present as of v31c425 — the current build adds it. The rest of this section documents the real, current UI.*

A field becomes calculated by setting its Type to **Number (integer)** and its **Calculated** toggle to **yes**. Once calculated, the field's own Entry-mode/preset controls disappear (they don't apply — nothing is ever typed into it directly) and a **Calculated** badge appears on the field card as a visual flag that this value comes from a formula, not user entry.

**Building the formula:** the formula input itself is deliberately read-only — you never type into it freehand. Instead you build it token by token with two dropdowns next to it: - **Insert field…** — lists every *compatible peer field* (another Number/integer field on the same form, including other calculated fields, so a calculated field can depend on another calculated field) - **Operator…** — +, −, ×, ÷, (, )

Each selection appends its token to the formula and the dropdown resets, so the formula is always built by explicit, auditable clicks rather than free text. A **Clear** button resets it. The field itself is excluded from its own Insert-field list, so you can't wire a field to reference itself.

**Dependency chains work as expected — CoD before WSJF, for example.** If Field B's formula references Field A, and Field C's formula references Field B, the evaluator resolves A before B before C. This is stated directly in the UI's own helper text: *"Field names are indexed names. Eval order: dependencies first (CoD before WSJF), then PEMDAS."*

**Locking:** once a calculated field's formula has been published (i.e., it's a persisted field on a form design already in use), the formula locks — you cannot edit it in place. To change a published formula, retire the field and add a new one. This is the same "retire, don't silently rewrite" discipline CEE applies to every other locked/published field attribute.

**Live preview on the record form:** while entering a record, calculated fields render read-only with a **"(calculated · live)"** label, and their displayed value recalculates on every keystroke in any field they depend on — a client-side PEMDAS evaluator (proper operator precedence and parenthesization, division-by-zero and malformed-formula caught as explicit errors rather than silently producing garbage) gives you an immediate preview. That said, **the preview is not authoritative.** The final value is

computed and locked in server-side at save time; the client evaluator exists purely so you can see the number before committing, not as the system of record.

**A few edges worth knowing:** - Calculated fields are excluded from CSV/PDF test-data generation (generatedRecordValues explicitly skips them) and from CSV import — you can't set a calculated field's value by import; it's always computed. - Switching a field's Type away from Number (integer) clears its Calculated flag and formula automatically (a formula only ever makes sense in a numeric field). - Switching a field's Type to **File attach** likewise clears any calculated/formula state — the two are mutually exclusive on a given field.

---

## 9. Reports and PDF Generation

**Where:** Import/Export menu → **Export Found Set** (the same dialog now handles both CSV and PDF — it's titled **Export / Report**).
*Same correction as §8: absent in v31c390, present in v31c425. FileMaker-parity PDF reporting has been added on top of the existing CSV export path rather than as a separate surface, so this section supersedes the "Export" half of §5's format options where relevant.*

Opening **Export / Report** now starts with a **Format** choice: **CSV (data interchange)** or **PDF Report**. Everything below Format — scope, field selection, ordering — is shared between both formats; only the output differs.

**Scope**, same three options as CSV export always had: current found set, tagged records (if any are tagged), or just the current record. The dialog defaults intelligently — if you have records tagged, it defaults to Tagged; otherwise, Current found set.

**Field selection:** the field list defaults to your form design's browse-grid columns, in their browse order, pre-checked; every other field on the form is listed below a divider, unchecked. Each field row has ↑/↓ to reorder it in the output, and there are **Select all** / **None** buttons for bulk toggling. An **Include canonical record ID** checkbox adds the record's system ID as a column if you want it.

| | |
|---|---|
| | |
| | **PDF-specific: layout mode.** Once Format is set to PDF, a **PDF layout** selector appears with two options: |

**Layout**
**What it produces**
**Tabular list**
One multi-record table, columns = your selected fields in your chosen order. Auto-switches to landscape orientation at 5+ columns, and shrinks font size progressively as column count grows (down to 6.5pt at 12+ columns) so wide schemas stay legible rather than clipping. Long cell values are ellipsized rather than overflowing. Repeats the header row and paginates automatically via jsPDF-AutoTable.
**Detail**

One record per page, form-style: field label on the left, value on the right, one row per selected field. Each page is headed with that record's display name. Best for something you'd actually hand someone as a document, versus a table for scanning many records at once.
Both layouts get a consistent header block (title, scope subtitle, operator/timestamp/CEE-version metadata line) and a footer with the Hub ID, job ID, and page number.

**Generation and governance:** clicking **Generate PDF Report** / **Generate Detail PDF** builds the document client-side (via jsPDF + the AutoTable plugin, loaded on demand from CDN — nothing is rendered server-side), computes a SHA-256 digest of the finished PDF bytes, and CU-attests the export job exactly like a CSV export does — same import-export-job artifact shape, with format: "pdf" and a report_kind of either tabular-list or detail-one-per-page. The **PDF Report Generated** confirmation shows record count, layout, scope, column count, the SHA-256 digest, job ID, and CU ID, so a generated report is just as auditable as any other governed export — it isn't a side-channel document dump.

The browser then downloads the file directly (<hub>-<scope>-<timestamp>.pdf, or with a -detail suffix for Detail layout) — there's no server round-trip for the file itself, only for the attestation.

**Relationship to §5 Import/Export:** this is genuinely the same dialog and the same governed job pipeline as CSV export, just with PDF added as a second output format. If you've read §5, everything about scope semantics, field selection, and CU attestation there carries over unchanged; the only new concepts are Format and PDF layout.

---

### 10. Governed Script Triggers (Trigger Definitions)

**Where:** the **Trigger** menu → **Browse Trigger Definitions**.
This is CEE's automation layer, and it's built on a hard architectural constraint that's worth understanding before you use it: **there is no free-form code surface, anywhere, ever.** Every action a Trigger Definition can take is one of a fixed set of allowlisted "governed executors." The save path itself pattern-matches the outgoing JSON for anything that looks like "javascript", "script", "eval", "shell", "command", or "code" as a key and hard-rejects the save if it finds one. This isn't a UI nicety you could route around from the API — it's enforced at the point of persistence.

**Trigger Definition browse.** Because a Hub may eventually carry multiple Trigger Definitions, this is a browse surface, not a single edit form: a table of Name / Event / Status / Priority / Action count, with **Executor Catalog** (lists every currently-allowlisted executor and its description), **View Chain**, **Enable/Disable**, and **Edit Selected** / **Create Trigger**.

| | |
|---|---|
| | |
| | |
| | |
| | |

| | |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | **The Trigger Definition editor**, opened from that browse, has: |

**Name**, **Status** (Draft / Active / Disabled)
**Event** — one of on-validate, on-commit, on-create, on-delete, on-restore, on-open
**Priority** — an integer; determines execution order relative to other triggers on the same event
**Failure policy** — *Reject operation* (a failing action blocks the whole record operation) or *Record failure and continue*
An ordered **Actions** list. Each action row is a dropdown over the full governed executor catalog, plus field-aware parameter controls that change based on which executor you pick (for example, validation.require@v1 gives you a field picker and a rejection message; field.set-literal@v1 gives you the field and the literal value to stamp in). Actions can be reordered (↑/↓) and removed, and execute strictly in list order after the triggering event has been attested.
**The current executor catalog** (field.* executors stamp or normalize a value; validation.* executors gate the operation):
**Executor**
**Effect**
field.set-literal@v1
Sets a field to a fixed literal value
field.set-timestamp@v1
Stamps the current timestamp
field.set-principal@v1
Stamps the acting principal (creating-principal auto-entry)
field.set-uuid@v1
Generates and stamps a UUID
field.set-sequence@v1
Stamps a monotonically increasing sequence value
field.normalize-uppercase@v1
Normalizes a field’s value to uppercase
field.normalize-lowercase@v1
Normalizes a field’s value to lowercase
validation.require@v1
Rejects the operation if a chosen field is missing, with a configurable message
validation.reject@v1
Unconditionally rejects the operation, with a configurable message (for gating on conditions expressed elsewhere)
Saving requires at least one action — an empty Trigger Definition is refused with “Add at least one governed action before publishing.” A successful save is CU-backed (“Trigger Definition published”), and

**View Chain** opens the same universal chain viewer used throughout CEE, so a Trigger Definition's history is auditable the same way a record's or a form design's is.

---

*End of Chapter 1. Chapters on Form Design internals, the CU/chain model, permissions and RBAC, and the Spine/Worker administrative surfaces are outside this chapter's scope and belong in later chapters of the manual.*

---

# Chapter 13: The Spine Editor

*Source: index.html's crcPolicy() and architecture_contract (Industry Configurator source — this is where the Spine Editor's role is defined, from the other tool's own boundary statement); personality-vatican-8level.json's crc_isolation_policy block. spine-editor.html itself (the tool's actual UI) is present in the broader source tree but has not yet been reviewed in depth for this chapter — everything below describes the Spine Editor's confirmed* role, *not yet its operational UI. This chapter should be substantially expanded once that review happens; treat it as a scope statement, not a how-to yet.*
The Spine Editor is the tool that turns a logical personality (Chapter 6) into a physically deployed spine — it is the hinge between "here is an industry's organizational shape" and "here are running worker instances" (Chapter 8).

## What the Spine Editor's job is, stated by the systems around it

No source reviewed so far documents the Spine Editor from its own perspective in detail — but its scope is stated with unusual precision by the *other* tools that hand work off to it. The Industry Configurator's architecture_contract lists everything deferred_to_spine_editor:
**Hub/table definitions per Level** — deciding what governed tables actually exist inside each logical Level (Chapter 4) the Configurator produced
**Spoke/row definitions per Hub** — the record schema inside each Hub, i.e., authoring the Form Design (Chapter 14) that a Hub's records will conform to
**Form designs** generally
**CRC physical Instance count** — literally computing how many worker instances (Chapter 8) a deployment needs
**Whole-Level-to-Instance packing** — deciding which logical Levels get physically hosted together on the same worker instance, and which get split apart
**Deployment bindings** — the addresses, ports, and routing that turn a packing decision into an actually-running set of processes
The spine_editor_role field embedded directly in personality artifacts states the physical-sizing half of this most plainly: *"compute physical Instance count and pack whole logical Levels into physical Instances using deployment-specific complexity/risk inputs."*

## Why this is a genuinely different job from the Configurator's

Chapter 6 draws this line from the Configurator's side; it's worth restating from the Spine Editor's side because it clarifies what makes this tool necessary at all rather than redundant with the Configurator. A personality is meant to be reusable across deployments of wildly different scale and risk tolerance — the same FINRA personality might seed a small regional broker-dealer's single-instance deployment or a large SIFI's twenty-instance deployment. The Configurator cannot make that physical-sizing decision, because it doesn't know (and by design, is forbidden from knowing — see its own forbidden_outputs list in Chapter 6) anything about the specific deployment's actual principal population, risk profile, or infrastructure. The Spine Editor is where those deployment-specific inputs finally enter the picture, and where Chapter 5's ISS formula — carried forward as inert policy through the entire Configurator stage — actually gets evaluated against real numbers for the first time.

## The formula in practice: what the Spine Editor is presumed to compute

Chapter 5 established that S* (optimal principals per isolation boundary) is population-independent — the same target surface size applies whether the deployment has 50 principals or 50,000. The practical consequence for the Spine Editor, inferred directly from that math but not yet confirmed against the tool's own UI: given a personality's logical Levels and a deployment's actual principal count/risk inputs, the Spine Editor's packing decision should be choosing **how many physical instances to stand up at the optimal S*, then assigning whole Levels to instances** — not shrinking or growing individual instances to fit an arbitrary Level count. Whether the tool's actual UI frames this as directly as "here is your computed S*, here is how many instances that implies" is unconfirmed; this is the mathematically consistent expectation given everything upstream of it, not a description of a screen anyone has actually looked at yet for this manual.

## Open items for this chapter

This chapter needs a real pass against spine-editor.html itself: what the packing UI actually looks like, whether attestation_boundary-flagged Levels (Chapter 4's Vatican example) get any special treatment during packing, how a packing decision is saved/published (presumably as a CU, per Chapter 1, the same way a Form Design publish is), and how this connects operationally to the Acceptance/Production promotion flow documented in Chapter 2. Until that review happens, this chapter states the Spine Editor's *contract* with confidence (it's directly sourced from two other tools' own boundary statements) but its *operation* only as a reasoned expectation.

# Chapter 14: Form Design

*Not yet drafted.*

Form Design is referenced throughout this manual as a settled concept — every Hub (Chapter 2) has one, Chapter 12 documents the field-level mechanics that live inside it (Required/Indexed/Case toggles, Calculated Fields, File attach fields), and Chapter 13 names authoring form designs as one of the Spine Editor's core responsibilities. What this manual doesn't yet have is a dedicated chapter walking through the Form Designer's own interface directly — field-type selection, layout, publish/lock semantics beyond what Chapter 12 §6 and §8 already cover incidentally, and how a form design's version history is presented.

This should be drafted from chandra-enterprise-editor.html's Form Designer screens directly, the same verification-first approach used for Chapter 12.

---

# Chapter 15: Permissions and RBAC — Chandra Passport

*Source: James, directly — confirming that "gr-identity," referenced in earlier drafts of this chapter and elsewhere in this manual, was an early internal working title for what is now called* ***Chandra Passport****. It is in active development and not yet available for beta. Cross-reference: Chapter 10 independently identified Chandra Passport's public-facing role (page-integrity/CU verification for chandrahub.net itself) from the site's own footer text — this chapter and that finding describe the same system from two different angles, now confirmed as one and the same rather than two separate things.*

This is the chapter that would resolve Chapter 1's most consequential open gap: request-level authentication and access control. As documented in Chapter 9, the CEE client sends no bearer token, API key, or other credential on any call — only an operator string, which is attribution for the CU chain (Chapter 1), not authentication. Nothing reviewed anywhere in this manual's source material — not the CEE client, not the source zip's Lisp files, not any of the live-fetched sites — describes how a request is actually authorized to act on a given spine.

## What's now confirmed

**Chandra Passport is Chandra's RBAC and identity-governance system.** It was developed under the internal working title "gr-identity" before being renamed — the two names refer to the same system, not two different ones, and any reference to "gr-identity" elsewhere in earlier drafts of this manual should be read as Chandra Passport under its earlier name. It is **in active development and not yet available for beta** — a real, in-progress system, not vaporware, but not yet something an operator can point at, install, or verify against.

**Chandra Passport appears to have two roles, seen from two different angles across this manual:**

**RBAC / principal-attribution substrate** (this chapter's original concern) — the mechanism that would close Chapter 1's authentication gap, presumably enforcing against Chapter 5's principal vocabulary (a distinct authority-bearing signing credential — human, agent, service account, API key, examiner, workflow engine, or integration identity).

**Public content integrity for chandrahub.net itself** (Chapter 10's finding) — the site's own footer states *"Page integrity: Pending chandrapassport deployment — CU verification active post-launch,"* meaning the examiner reference and other chandrahub.net pages are intended to eventually be CU-attested artifacts in their own right, not just static pages describing CU-attested artifacts elsewhere.

Whether these are genuinely one unified system serving both purposes, or two related-but-distinct capabilities that happen to share a name, hasn't been confirmed — worth asking directly once Chandra Passport is further along, rather than assuming either way.

## What this chapter still doesn't cover

No real source (UI, API, or backend code) has been reviewed for Chandra Passport — everything above is confirmed *positioning* (what it's called, what gap it's meant to close, that it's in development), not confirmed *implementation*. This chapter should be rebuilt from real source the same way every other chapter in this manual was, once Chandra Passport reaches beta and there's something concrete to verify against. Until then, don't draft operational guidance here based on assumption — a name and a stated purpose are not a specification.

# Chapter 16: Marshaller Administration

*Not yet drafted.*

Chapter 7 documents the Marshaller's core registry function — address-only persistence, VERIFY semantics, GR Marshaller vs. Client Marshaller, and the Attestation Console. What's still needed here is the *administrative* layer on top of that: how a Marshaller instance itself is deployed and configured (is it just another worker instance per Chapter 8's model, or does it have its own distinct deployment shape?), how its own acache (Chapter 18) is backed up or migrated independently of the spines it tracks, and how the Attestation Console's relationship to chandrahub.net's examiner-facing attestation model (flagged as unconfirmed in both Chapters 7 and 10) actually works.

This chapter should be drafted once that relationship is confirmed directly, and once chandra-marshaller.html's administrative screens (as opposed to its day-to-day registry-browsing screens, already covered in Chapter 7) have been reviewed specifically for setup/config/backup concerns.

---

# Chapter 17: Worker Deployment and Diagnostics

*Not yet drafted.*
Chapter 8 covers the conceptual architecture of worker instances — one process, one database, one isolation surface, and the documented AllegroCache close/reopen incident worth avoiding. What's still needed is the operational half: the actual deployment scripts seen in the source zip but not yet walked through (run-all-ent.sh, restart-all-ent.sh, status-all-ent.sh, stop-ent.sh, the systemd install script ac-install-chandra-systemd.sh), what a healthy vs. unhealthy fleet looks like across chandra-worker-diagnostics.html's full surface (not just the identity-manifest piece already covered in Chapter 8), and how a new worker instance is actually provisioned end-to-end — from a Spine Editor packing decision (Chapter 13) to a running, registered (Chapter 7) instance.

This chapter is a strong candidate for the next drafting pass, since most of its raw material (the shell scripts, the full Worker Diagnostics tool) is already sitting in reviewed source and just hasn't been walked through operationally yet.

---

# Chapter 18: Help Content Architecture — Per-Surface ACache Isolation

*Source: cee-instance-dashboard-help-batch-v1.json and marshaller-help-batch-v1.json (real help-batch artifacts, both generated_at: 2026-08-23T13:32:00-05:00); marshaller.chandrahub.net (live-fetched, confirms the Help Index UI labeled "LOCAL MARSHALLER ACACHE"); ent-help-context-authority.lisp, ent-environment-roots.lisp (source zip); James, directly, on the Mythos graph-edge rationale below.*

Chandra's in-product Help system is deliberately **not** a single, shared content store serving every tool. It is split by **authority** — a concrete, shipped mechanism, not just an architectural intention — and understanding why is a direct, worked application of Chapter 5's CRC argument to a piece of the system that might otherwise look like an obvious candidate for consolidation.

## The mechanism: chandra-help-batch-v1 artifacts

Help content ships as versioned batch artifacts, each stamped with an explicit authority:
**authority: "spine-acache"**, surface_group: "cee+instance-dashboard" — 41 contexts, version: "v31c452". CEE (Chapter 2) and the per-worker Instance Dashboard (Chapter 8) share this one help authority.
**authority: "marshaller-acache"**, surface_group: "marshaller+fleet-dashboard" — 37 contexts, version: "v43". The Marshaller (Chapter 7) and its Fleet Dashboard share a separate authority.
Each Help context carries a context_id, title, scope, default_body, tags, a status, and a context_type. The Marshaller batch's own marshaller-overview context states the isolation boundary directly, in the product's own words, to anyone who opens it: *"Marshaller is the fleet-level discovery and observation surface. Its persistent registry authority is local to the Marshaller ACACHE. It does not read or write CEE or worker Help Contexts."* This isn't a design document's aspiration — it's live help text a Marshaller operator can read today, confirmed directly on marshaller.chandrahub.net, whose Help Index UI labels its own source plainly: **"LOCAL MARSHALLER ACACHE."**

## The merge policy: safe, idempotent reseeding

Both batches carry the same merge_policy:
existing: upsert-standard-preserve-authored
preserve: [user_body, title, tags_json]
update: [default_body, scope, status, context_type,
form_design_id, field_id, field_label, level_id, hub_id]
unchanged: skip-no-cu
Reseeding help content is safe to run repeatedly: system-owned fields (the default body text, scope, status, and field/form/level/hub linkage) get refreshed to whatever the new batch says, while anything a human operator has actually authored (user_body, a custom title, tags_json) is preserved rather than clobbered. And critically, per Chapter 1's governance principle — a context that hasn't actually changed is skipped **with no CU written at all** (skip-no-cu). Reseeding the same batch twice doesn't spam the chain with no-op attestations; only genuine changes produce new governed history.

## Why this exists — the actual thesis, stated directly by James

Keeping system Help monolithic — one shared store serving every tool — would open a graph edge a Mythos-class adversary could traverse. This is Chapter 5's Dunbar Perimeter argument, applied to Help content specifically rather than business data: a single Help store touched by CEE, the Marshaller, every worker's Instance Dashboard, and anything else in the ecosystem is exactly the kind of unowned, cross-

surface connection CRC's Consolidate pillar exists to eliminate. It doesn't matter that Help content feels lower-stakes than business records — a traversable graph node is a traversable graph node regardless of what's sitting in it, and an unbounded adversary doesn't care that the content in question is "just help text."

**This matters more than it might look, because Help content is client-customizable for a client's own infosec needs.** A shared, monolithic Help store would mean one client's customized Help content — which might describe that client's own specific procedures, terminology, or internal security posture — sits in the same traversable graph node as every other client's customizations and every other surface's content. Splitting by authority means a compromise or traversal reaching CEE's Help content structurally cannot reach the Marshaller's, and vice versa, regardless of how the underlying infrastructure is otherwise networked.

## The same posture also serves disaster recovery

Chapter 10 makes this same dual-argument about organizational data generally: isolating content down to fine-grained, independently-scoped segments is not purely a security measure. It's also a disaster-recovery one — a corruption or loss affecting one authority's acache doesn't cascade into every other surface's Help content. This is the same underlying design instinct (isolate everything down to the smallest sensible segment) serving two related but distinct purposes at once, and it's worth reading Chapters 10 and 18 together for that reason: neither chapter's argument is complete on its own.

## Open question, not yet resolved

The as-shipped split is **two** authorities (Spine-acache, serving CEE + Instance Dashboard; Marshaller-acache, serving Marshaller + Fleet Dashboard) — not literally "one acache per worker instance" as a third category, which is how the original architectural bookmark for this chapter was phrased. Whether the current two-authority split is the intended final shape, or whether a further per-worker-instance split is still planned on top of it, has not been confirmed and should be resolved with James directly before this chapter is treated as describing a finished architecture.

# Chapter 19: AegisGenera / Reference-Monitor Gateway

*Confirmed unbuilt as of this manual's drafting — per James, directly, and consistent with a direct search of the full source zip, which contains no AegisGenera source anywhere. aegisgenera.com is referenced in CRC's own site navigation (Chapter 5) but has not itself been fetched or reviewed.*

AegisGenera is referenced elsewhere as a "hardened reference monitor gateway" and vendor-agnostic compliance middleware. Two things in this manual's already-verified source material point at what its eventual role is likely to be, though neither is a confirmation:

1. **CRC's own Pillar 04 (Boundary) description** (Chapter 5) names exactly the capabilities a reference-monitor gateway would need to implement: certificate-pinned endpoints, request signing, anomaly detection, circuit breakers, a governed endpoint registry, and GABA attestation (Chapter 11) for computer-use agents specifically. This is a strong structural fit for what AegisGenera is described elsewhere as being — but CRC's homepage doesn't actually name AegisGenera as the Pillar 04 implementation the way it explicitly names DXMachine for Consolidate and "Chandra Protocol + DXMachine" for Reduce/Close. That specific attribution is inferred, not stated.
2. **Chapter 10's positioning of AegisGenera as the actual end-state client offer** — per James directly, the real answer to "how does a client get control and security" is a sovereign, self-hosted deployment running their own AegisGenera boundary, not a seat on shared chandrahub infrastructure. This tells us AegisGenera's *commercial role* (the thing a client ultimately runs to be sovereign) even though it doesn't yet tell us its *technical* one in detail.

Nothing in this manual's reviewed source describes AegisGenera's actual architecture, request flow, or how it would integrate with a worker instance's own network boundary (Chapter 8) or the auth gap flagged in Chapter 15. This chapter should be drafted once real AegisGenera source or documentation exists, using the same verify-before-writing discipline as every other chapter in this manual — not extrapolated further from the two adjacent-but-indirect references above.

# Chapter 20: The CAMM Framework

*Source: chandrahub.net/docs/standards.html (uploaded copy, 2026-08-24). This is a genuinely thin chapter — one paragraph of source material exists, no whitepaper has been reviewed yet despite one being referenced. Placement note: this chapter is appended at the end of the manual (Part III) purely for build-order convenience — conceptually it belongs alongside Chapter 5 (CRC) and Chapter 11 (GABA) in Part I, as a third standard in the same family. A future revision should renumber it into Part I rather than leaving it stranded here; this placement is provisional, not a statement that it belongs in Administration.*

**CAMM** — the **Chandra Audit Maturity Model** — is a third standard alongside CRC (Chapter 5) and GABA (Chapter 11), distinct in what it measures: where CRC scores a *deployment's* architectural posture (0–16 across four pillars) and GABA attests *AI inference/computer-use boundaries* specifically, CAMM scores an *organization's* maturity in adopting Chandra as its audit infrastructure — a different axis entirely, closer to a capability-maturity model than a deployment scorecard.

## What's confirmed

CAMM defines **five maturity levels**, described as running "from basic CU production through full forest federation with cross-instance trust." That's the entire confirmed description available as of this chapter's drafting — the specific criteria for each of the five levels, how an organization is assessed or certified against them, and who administers that assessment are all undocumented in what's been reviewed so far.

A whitepaper is referenced (linked from chandrahub.net's Standards page) but has not itself been fetched or reviewed for this manual. This chapter should be substantially rewritten once that whitepaper exists to draw from directly — right now it states that CAMM exists and roughly what it measures, nothing more.

## Where it likely connects

Speculative, not confirmed: "full forest federation with cross-instance trust" as the top maturity level almost certainly ties to Chapter 10's chandrahub/forest concept and the "Cross-Forest Trust Protocol" listed as forthcoming on the same Standards page — and possibly to Tickler (Chapter 1's fifth primitive), which is itself described as "the federation primitive." If that connection holds, CAMM's top level may specifically be measuring an organization's use of Tickler-based cross-forest mechanisms rather than federation in some more general sense — but this is inference stacked on inference from thin source, not something to state as fact anywhere else in this manual until real source confirms it either way.

## What this chapter still needs

The actual Level 1–5 criteria; who assesses/certifies against CAMM (self-assessment, examiner-administered, General Reasoning-administered); whether a CAMM level is a prerequisite or complement to CRC certification (Chapter 5) or GABA attestation (Chapter 11); and the whitepaper itself, reviewed directly rather than summarized from a one-paragraph blurb.