The Parmana Handbook

Every capability, explained by reading the actual source.

Generated 2026-09-16

The Parmana Handbook

Every capability this codebase has, no matter how small, explained by reading the actual source rather than trusting existing documentation. This is a new document, written 2026-09-16, independent of the older `docs/book/` (dated 2026-09-05, not updated here) and independent of the frozen root-level architecture documents from mid-2026. Where this book disagrees with either, this book was written later and checked against the code that exists today; say so if you find a place where that turns out not to be true.

How each chapter is structured

Every chapter answers the same six questions, in the same order:

1. **What it is.** One paragraph, plain language.
2. **Why it was built.** The real problem, usually quoted from the source's own comments.
3. **How it works.** Traced to real files, usually with line numbers.
4. **How it enables things, with an example.** A real tutorial or test that proves the behavior, or an honest note that none exists yet.
5. **How to validate it yourself.** The exact files to open.
6. **Integration requirements**, where applicable: env vars, config, external dependencies.

Chapters

#	Title
23	History and Open Questions

Corrections found while writing this book

Every chapter above was written by reading the actual current source, not by trusting existing documentation, tutorial comments, or `.env.example`. That process surfaced real places where something written down elsewhere in this repository no longer matches the code. Listed here, in one place, rather than only buried inside the chapter that happened to find each one:

- **`.env.example` claims `KEY_PROVIDER=aws-kms` "does nothing."** False. `SignerBootstrap.ts` implements it for real, via `KmsSigner`. See Chapter 2 and Chapter 3.
- **A live `.env` sets `KMS_REGION` and `KMS_KEY_ALIAS`.** Neither is read anywhere in `packages/*/src`. The real environment variables `KmsSigner.ts` actually reads are `AWS_REGION` and `AWS_ROLE_ARN`. Confirmed independently by two separate chapters (2/3 and 20/21) reading the source directly. See Chapter 2 and Chapter 21.
- **`KeyBootstrap` is effectively legacy.** Its own sibling, `SignerBootstrap.ts`, states in its own comment that nothing in this codebase's production paths actually calls `KeyBootstrap` today. See Chapter 2.
- **A prior architecture assumption placed `ExecutionGateway` at `packages/api/src/execution-gateway/ExecutionGateway.ts`.** That path does not exist. The real location is `packages/execution-gateway/src/ExecutionGateway.ts`, a separate top-level package. See Chapter 10.
- **An older claim that `@parmana/replay` and `@parmana/receipt` are both unwired, tutorial-only packages is only half true.** `@parmana/replay` is confirmed genuinely unimported by `packages/api/src`. `@parmana/receipt` does not exist as a standalone package at all; receipt functionality is real and production-wired inside `@parmana/crypto` (`ReceiptCrypto` / `ReceiptHasher`). See Chapter 18.
- **`SupabasePolicyRepository` sits outside the `StorageProvider` facade** every other Supabase-backed repository in this codebase uses. This is intentional, not an oversight: `PolicyRepository` is an interface owned by `@parmana/policy`, not `@parmana/shared` / `@parmana/storage`, and predates the governance/storage split by design. Noted here so a reader comparing Chapter 13 against `StorageProvider.ts` doesn't mistake the asymmetry for a bug.
- **The rate-limiter store-reuse incident (`examples/tutorials/115`) has a documented, corrected misdiagnosis worth repeating here**, since it's a real lesson about not trusting a log line's proximity to a failure as proof of causation: the observed HTTP 500 was originally blamed on the `ERR_ERL_STORE_REUSE` warning; the real cause was the unrelated `SignerKeyProviderAdapter` signing/verification key divergence (G-48/G-49) logged in the same request. See Chapter 16 and Chapter 23.

Other formats

- **Docs site (Mintlify):** the same chapters, published at `docs/site/handbook/`, discoverable from the main navigation.
- **PDF:** a single-file download, gated behind an email address (no verification email sent, see the download page for why). Generated from these same chapter files by `scripts/generate-handbook-pdf.ts`.

What this book is not

It is not `docs/CLAIMS.md` (the audit ledger, one entry per claim with cited evidence) or `docs/VERIFICATION-GAPS.md` (the dated incident and gap log). Read those for the history of a specific claim or gap. This book is the narrative connecting the pieces, written for someone who needs to build on, audit, or extend this codebase, or integrate against it for the first time.

Chapter 1: The Trust Model and Domain Model

What it is

Parmana models every request that could result in a real-world action as a chain of six immutable artifacts: Authority, Authorization, Intent, a Decision produced by evaluating that Intent against a Policy, an Execution that records what actually happened, and an Execution Trust Record that binds all of it into one signed, append-only document. Nothing in this chain is ever mutated after creation. A correction is a new artifact, never an edit to an old one.

Why it was built

`packages/shared/src/domain/business-transaction.ts` states the design directly in its own doc comment: a `BusinessTransaction` "captures the complete upstream trust chain prior to policy evaluation," and the chain itself is drawn as a diagram right in the source:

```
Authority
     ↓
Authorization
     ↓
Intent
     ↓
BusinessTransaction
     ↓
PolicyReference
```

The reasoning is that a policy decision is only meaningful if you can also prove who asked for it, under what authorization, and exactly what they asked for, all as facts that existed before the decision was made and cannot be edited afterward to match whatever the decision turned out to be. `intent.ts` restates the same chain with `Decision` and `Execution` added on the end, and is explicit that "Intent is evaluated by Policy but is never modified by Policy" and that `decision.ts` "does not create authority... does not grant authorization... does not modify intent." Each artifact in the chain has exactly one job and cannot reach backward to change an earlier one.

How it works

The six domain types

All defined in `packages/shared/src/domain/`, one file per type:

`Authority` (`authority.ts`), the entity behind a request. Has an `authorityId`, an `authorityType` (`AuthorityType.USER | ROLE | SERVICE | ORGANIZATION`), a `principalId`, and an `issuedAt` timestamp. Nothing else. There is no field anywhere on this type, or on anything downstream of it, that says "this authority is an AI" or "this authority is a human" in a way policy evaluation can see.

`Authorization` (`authorization.ts`), proof that an `Authority` approved an intended execution, with an `authorityId`, a `purpose` string, `issuedAt`, and an optional `expiresAt`.

`Intent` (`intent.ts`), the actual business action being requested: `action` (e.g. `"TransferFunds"`), `target` (e.g. `"account/12345"`), and free-form `parameters`.

`BusinessTransaction` (`business-transaction.ts`), the aggregate that wraps `Authority`, `Authorization`, `Intent`, a `PolicyReference`, and the runtime `signals` a policy will evaluate. It carries its own lifecycle (`BusinessTransactionStatus`: `RECEIVED → POLICY_EVALUATED → APPROVED | REJECTED → (OVERRIDDEN) → EXECUTING → EXECUTED | FAILED → VERIFIED`). This is the immutable input `RuntimeEngine.execute()` receives.

`Decision` (`decision.ts`), the pure output of evaluating an `Intent`'s signals against a `Policy`: an `outcome` (`APPROVED | REJECTED`), the `matchedRuleId`, `evaluatedRules` (count), and `matchedPath` (the ordered rule-id trace `PolicyEngine` walked). The last three fields are optional only because a `Decision` built before they existed (see `docs/VERIFICATION-GAPS.md` G-44)

doesn't carry them retroactively; every decision going forward does. A `Decision` never carries anything the caller supplied that wasn't independently produced by evaluation itself.

`Execution` (`execution.ts`), what actually happened while processing the transaction: wraps exactly one `Decision`, has its own `ExecutionStatus` (`PROCESSING | COMPLETED | FAILED`) and `ExecutionMode` (`SYNC | ASYNC`), and optionally carries a hash-chained `previousChainHash` / `chainHash` / `chainSignature` linking it to the prior `Execution` for the same transaction (see `ExecutionChainCrypto`, Chapter 3).

`ExecutionTrustRecord` (`execution-trust-record.ts`), the aggregate root: one `BusinessTransaction`, arrays of `overrides` / `executions` / `verifications` / `receipts` (arrays, not single values, because a long-running or retried transaction can accumulate more than one of each), an optional signed `authorization` (the exact `SignedExecutionAuthorization` the Execution Gateway accepted, see Chapter 7), a `trustRecordHash`, and a `signature`. This is "the authoritative source for replay, verification, audit, and receipt generation," per its own doc comment, and is itself signed as a whole so that any single field being altered after the fact invalidates the record.

PolicyReference: the field that proves which policy content, not just which version

`policy-reference.ts` carries `name`, `version`, `schemaVersion`, and two fields worth understanding closely: `contentHash` and `governanceAnchor`. Both are marked explicitly as "caller-unsettable" in the source comments, a request-supplied `PolicyReference` never carries either; `RuntimeEngine` computes both, after loading the real policy document, and merges them only into the copy embedded in the trust record's `transaction.policy`, never into the transaction as originally submitted. `contentHash` is a sha256 of the canonicalized policy content actually loaded (closing the gap where an in-place edit to an existing version's `policy.json` would otherwise be undetectable from the trust record alone, `docs/VERIFICATION-GAPS.md` G-24). `governanceAnchor` records whether that content is traceable to a completed Policy Governance approval (`VERIFIED | NO_APPROVAL_RECORD | SIGNATURE_INVALID | CONTENT_MISMATCH`), resolved unconditionally regardless of whether execution-time enforcement of that fact is turned on (see Chapter 14).

EvidenceAnchor: the explicit "these three things are linked" pointer

`evidence-anchor.ts` is newer and narrower: a single computed `anchorHash` over `{policyContentHash, governanceAnchorStatus, connectorEvidenceHash}`. Its own doc comment is unusually candid about what it is not: "Not a NEW cryptographic guarantee... What this adds is a single, explicitly-named, independently-computed pointer an auditor can check WITHOUT already knowing to reach into `transaction.policy` and `executions[].evidence.attributes.connector` separately and reconstruct the binding themselves." It exists because an audit (`docs/investigations/2026-09-15-evidence-anchor-gap-audit.md`, GAP-4) asked whether policy governance provenance and actual connector evidence were provably linked, and the honest answer at the time was "implicitly, but nothing says so directly."

Caller-type agnosticism

Nowhere in `packages/runtime/src`, `packages/policy/src`, or `packages/execution-gateway/src` does any evaluation or gateway-release logic branch on `AuthorityType`. A search across all three packages for `AuthorityType.` outside test files returns nothing. The only place `credentialHolderType` / `AuthorityType` is read for a real decision is at the HTTP caller-auth layer, specifically `isHumanCaller()` for Policy Governance approve/reject (Chapter 14), a narrow, deliberate exception, not evidence that the core trust chain treats callers differently. An AI agent, a human, a service account, and an organization-level credential all produce identical `Authority` / `Intent` / `Decision` / `Execution` shapes and go through the exact same `RuntimeEngine.execute()` path.

How it enables things, with a concrete example

`examples/tutorials/100-authorization-caller-type-agnostic` and `packages/api/tests/integration/authority-type-agnostic-execution.integration.test.ts` both exist specifically to prove this: the integration test's own title is "produces an identical APPROVE decision regardless of authority.authorityType, including a value outside the AuthorityType enum entirely," and a second case proves an identical REJECT decision (same `matchedRuleId`, same reason shape) across authority types too. This is the concrete evidence behind the "caller-type-agnostic" claim above, not just an absence-of-special-casing argument.

More broadly, the domain model is what every other capability in this book builds on: Chapter 2's `RuntimeEngine.execute()` walkthrough operates entirely in terms of these types; Chapter 5's runtime pipeline is the process

that turns a `BusinessTransaction` into an `ExecutionTrustRecord`; Chapter 6's cryptography chapter explains exactly how `trustRecordHash` / `signature` get computed over this aggregate.

How to validate this yourself

- `packages/shared/src/domain/business-transaction.ts`, `authority.ts`, `authorization.ts`, `intent.ts`, `decision.ts`, `execution.ts`, `execution-trust-record.ts`, `policy-reference.ts`, `evidence-anchor.ts`, the type definitions themselves, each with a doc comment explaining its own role in the chain.
- `packages/api/tests/integration/authority-type-agnostic-execution.integration.test.ts`, proves caller-type agnosticism at the HTTP boundary.
- `examples/tutorials/100-authorization-caller-type-agnostic/run.ts`, the same proof, as a runnable narrative.
- `grep -rn "AuthorityType\." packages/runtime/src packages/policy/src packages/execution-gateway/src` (excluding test files), reproduces the "nowhere is this special-cased" claim directly.

Integration requirements

None specific to the domain model itself, these are plain TypeScript interfaces with no external dependency. What consumes them (the runtime pipeline, storage, crypto) each carries its own requirements, covered in later chapters. The one practical requirement worth naming here: any caller submitting a `BusinessTransaction` must supply a well-formed `Intent` (`action`, `target`, `parameters`) and a `PolicyReference` naming a real, loadable policy, `contentHash` and `governanceAnchor` must never be supplied by the caller, since the server computes and overwrites both regardless of what's submitted.

Chapter 2: Configuration and Bootstrapping

What it is

Every runtime-configurable choice Parmana makes, which storage backend, which signature algorithm, where keys live, whether caller authentication is on, flows through exactly one function: `loadConfig()` in `packages/shared/src/config/Config.ts`. Everything downstream, from which `Signer` gets constructed to whether the process refuses to start at all, is derived from the single immutable object that function returns.

Why it was built

`Config.ts`'s own doc comment states the goal plainly: "Centralized immutable configuration. This is the only configuration model used by Parmana." The practical reason is fail-closed startup. A process that boots successfully, passes `/health`, and only fails on its first real request (because a key file is missing, or a database is unreachable) is worse than a process that refuses to start at all. Several files across `packages/api/src/bootstrap/` exist specifically to convert what would otherwise be a lazy, request-time failure into an eager, startup-time one, and their own comments say so directly: `assertSigningKeyMaterialConfigured.ts` opens with "Without this, `FileKeyProvider` only throws lazily... A production process could otherwise boot 'successfully', pass `/health`, and only fail on its very first real request."

How it works

`loadConfig()`, section by section

`loadConfig()` (`Config.ts:305-381`) reads `process.env` exactly once per call and returns a frozen `Config` object with eleven sections: `environment`, `storage`, `crypto`, `keys`, `secrets`, `authorization`, `policy`, `trust`, `api`, `auth`, `rateLimit`, `logging`. Every value is parsed and validated through `ConfigValidation.ts`'s `parse*` functions, each of which throws immediately on an unrecognized value rather than silently falling back to a default. Two sections fail closed even harder, by refusing an *unset* value rather than just an invalid one: `requirePolicyDirectory()` throws if `PARMANA_POLICY_DIR` is unset, and `assertSigningKeyMaterialConfigured.ts` (a separate, startup-only check, not inside `loadConfig()` itself) does the same for `PARMANA_KEY_DIR`.

`.env` resolution is location-independent: `findEnvFile()` walks up from `Config.ts`'s own directory until it finds a `.env` file, so every package in the monorepo shares the same configuration regardless of which package's `cwd` a script happens to run from.

The full environment variable reference

Every variable below is read somewhere in this monorepo; `.env.example` is the canonical list (351 lines, heavily commented) and was the primary source for this table, cross-checked against `Config.ts` / `ConfigValidation.ts` for the ones that carry a real enum.

Variable	Required?	Valid values	Default	What it controls
`NODE_ENV`	No	free-form; only `"test"` changes behavior anywhere	`development`	Test-mode bypasses (memory storage, skipped startup asserts)
`PARMANA_POLICY_DIR`	**Yes**	any directory path	,	Where `policy.json` files live
`PARMANA_KEY_DIR`	Yes unless `NODE_ENV=test`	any directory path	,	Where signing key `.pem` files live
`PARMANA_KEY_MATERIAL_JSON`	No	JSON object, see below	unset	Materializes key files from env on platforms with no

Variable	Required?	Valid values	Default	What it controls
				volume/secret-file primitive
`PARMANA_GATEWAY_KEY_ID`	No	any string	`gateway`	Key-file prefix for the Gateway's attestation keypair
`PARMANA_GATEWAY_ID`	No	`[A-Za-z0-9._-]+`	`parmana-gateway`	Logical Gateway identity name in audit events
`PORT`	No	number	`3000`	HTTP bind port
`HOST`	No	any	`0.0.0.0`	HTTP bind interface
`SHUTDOWN_TIMEOUT_MS`	No	number	`10000`	Graceful-shutdown drain window
`LOG_LEVEL`	No	free-form	`info`	Log verbosity string (no enforcing logger library)
`PARMANA_STORAGE`	No	`memory` | `postgres` | `supabase`	`memory`	Storage backend for business transactions/trust records
`DATABASE_URL`	Yes unless `NODE_ENV=test`	Postgres connection string	,	Direct `pg` connection, used regardless of `PARMANA_STORAGE` by nonce/audit stores
`SUPABASE_URL`	Yes if `PARMANA_STORAGE=supabase`	URL	,	Supabase project URL
`SUPABASE_SERVICE_ROLE_KEY`	Yes if `PARMANA_STORAGE=supabase`	`sb_secret_*`	,	Privileged service-role key
`SUPABASE_ANON_KEY`	Yes if `PARMANA_STORAGE=supabase`	`sb_publishable_*`	,	Low-privilege anon key (used by `scripts/verify-policy-changes-approved.ts`, Chapter 14)
`CRYPTO_MODE`	No	`single` | `hybrid` | `pq`	`single`	Whether a secondary signature algorithm also signs every trust record/receipt
`PRIMARY_SIGNATURE_PROVIDER`	No	`ed25519` | `ecdsa-p256` | `dilithium3` | `dilithium5` | `sphincs-plus` (also accepts `ml-dsa-65` as an alias for `dilithium3`)	`ed25519`	Primary signature algorithm
`SECONDARY_SIGNATURE_PROVIDER`	Yes if `CRYPTO_MODE=hybrid`	same list as above	unset	Secondary signature algorithm; must differ from primary
`HYBRID_SIGNATURE_REQUIRED`	No	`"true"` (strict equality)	`false`	Rejects a record whose `signatures` array

Variable	Required?	Valid values	Default	What it controls
				is absent/partial once hybrid mode is trusted
`HASH_PROVIDER`	No	`sha256 | sha3-512 | blake3`	`sha256`	Hash algorithm
`KEY_PROVIDER`	No	`local | aws-kms | azure-key-vault | gcp-kms | hsm`	`local`	Signing-key custody backend, see the note below, `local` and `aws-kms` are real; the other three are reserved
`AWS_REGION`	Yes if `KEY_PROVIDER=aws-kms`	AWS region string	,	Region for the `KMSClient`
`AWS_ROLE_ARN`	No	AWS role ARN	unset	If set, exchanges Vercel's OIDC token for short-lived STS credentials instead of the SDK's default credential chain
`EXECUTION_AUTHORIZATION_TTL_SECONDS`	No	number	`120`	Default TTL on a signed Execution Authorization
`TRUST_PROFILE`	No	`v1` (only value defined)	`v1`	Trust record schema profile
`RECEIPT_VERSION`	No	free-form	`1`	Receipt schema version string
`PARMANA_API_KEYS`	Yes unless `PARMANA_AUTH_DISABLED=true`	JSON array, see Chapter 13	`[]`	Caller identities and their key hashes
`PARMANA_AUTH_DISABLED`	No	`"true"` (strict equality)	`false`	Bypasses caller authentication entirely, local development only
`RATE_LIMIT_EXECUTE_PER_MINUTE`	No	number	`30`	`/execute`, keyed by caller identity
`RATE_LIMIT_HEALTH_PER_MINUTE`	No	number	`300`	`/health`, `/ready`, keyed by IP
`HUBSPOT_PRIVATE_APP_TOKEN`	No	HubSpot token	unset	HubSpot connector; connector simply isn't registered if unset
`GITHUB_APP_ID` / `GITHUB_INSTALLATION_ID` / `GITHUB_APP_PRIVATE_KEY`	No, but all-or-nothing	,	unset	GitHub connector
`PAYTM_CONNECTOR_URL` / `PAYTM_CONNECTOR_SHARED_SECRET`	No, but all-or-nothing	HTTPS URL / string	unset	Paytm connector (forwards to a separate trusted service, never calls Paytm directly)
`PARMANA_SECRETS_PROVIDER`	No	`env | aws-secrets-manager`	`env`	Where connector *bearer* credentials resolve from

Variable	Required?	Valid values	Default	What it controls
`DATABASE_PROVIDER`	**Never set**	,	,	Retired; setting it throws at startup naming `PARMANA_STORAGE` as the replacement

A genuine discrepancy worth flagging directly: `.env.example`'s own comment on `KEY_PROVIDER` says "Only `local` (FileKeyProvider) has an implementing class today, setting this to any of the other four values does nothing." That statement is now wrong. `SignerBootstrap.ts` (Chapter 3) implements `aws-kms` for real, via `KmsSigner`. The comment predates that work and was not updated afterward, exactly the kind of drift this book exists to catch rather than repeat.

`*Bootstrap` classes: composition roots, not dependency injection

Three classes in `packages/crypto/src/` follow an identical shape: a static `create()` method, a private static cache, construction driven entirely by `loadConfig()`.

- **`CryptoBootstrap`** (`CryptoBootstrap.ts`) builds a `CryptoProvider` (hash + signature) for a given algorithm, registering built-in providers (`SHA256HashProvider`, `Ed25519SignatureProvider`, `Dilithium3SignatureProvider`) into `HashRegistry` / `SignatureRegistry` and selecting by `config.crypto.hashProvider` / `primarySignatureProvider`. `createHybrid()` builds both primary and secondary providers at once for `CRYPTO_MODE=hybrid`.
- **`KeyBootstrap`** (`KeyBootstrap.ts`) is the older `KeyProvider` composition root. Its own doc comment is unusually direct about its current status: `KEY_PROVIDER` accepts five values for forward compatibility, but "only `FileKeyProvider` (`local`) is actually implemented," and it throws loudly for anything else rather than silently falling back to file-based keys, closing a real prior gap where a misconfigured `aws-kms` value used to parse cleanly and then quietly construct a `FileKeyProvider` anyway.
- **`SignerBootstrap`** (`SignerBootstrap.ts`) is the newer, signing-capable sibling (Chapter 3 covers `Signer` vs `KeyProvider` in full). Unlike `KeyBootstrap`, it supports both `local` (`LocalFileSigner`) and `aws-kms` (`KmsSigner`) for real, and is deliberately **not** memoized the way `KeyBootstrap.create()` is, a static singleton here previously poisoned test isolation, since each test sets its own `PARMANA_KEY_DIR` and a cached signer would keep pointing at a deleted temp directory across tests.

Startup order, traced from `server.ts`

`packages/api/src/server.ts` runs, in this exact order:

1. `assertStorageConfigured()`, refuses to start with `PARMANA_STORAGE=supabase` and no `DATABASE_URL`.
2. `assertSigningKeyMaterialConfigured()`, materializes `PARMANA_KEY_MATERIAL_JSON` into `PARMANA_KEY_DIR` for any file not already present, then confirms `default.private.pem` / `default.public.pem` exist (skipped for `KEY_PROVIDER=aws-kms`, which has no local file for that key by design).
3. `await assertKmsSigningKeyReachable()`, for `KEY_PROVIDER=aws-kms` only: constructs a real `SignerBootstrap`-backed signer and confirms the configured KMS key actually exists and is reachable, before binding the port.
4. `assertPaytmConnectorConfigured()`, refuses to start with only one of `PAYTM_CONNECTOR_URL` / `PAYTM_CONNECTOR_SHARED_SECRET` set.
5. `createExecutionSystem()`, `createApplication()`, `createCallerAuthenticator()`, `createRateLimitStore()` (twice, once per limiter, see the tutorial 115 case study in Chapter 12), the actual application graph.
6. `app.listen(PORT, HOST)`, the port only binds after every check above has passed.
7. `runPolicyGovernanceIntegrityCheckAtStartup()` and `schedulePolicyGovernanceIntegrityCheck()`, fired *after* the port is bound, deliberately fail-open (Chapter 14), these must never delay traffic from being accepted, unlike steps 1-4.
8. `createGracefulShutdown()` wired to `SIGTERM` / `SIGINT`.

The split between steps 1-4 (fail-closed, before the port binds) and step 7 (fail-open, after) is a deliberate, named discipline, not an accident of ordering, durable storage and signing key material are things a process cannot function without at all, while a Policy Governance integrity mismatch is something to detect and log, not something that should keep a healthy process from serving traffic.

How it enables things, with a concrete example

Every tutorial in `examples/tutorials/` that spins up a real `createApplication()` instance (the large majority of them) depends on `loadConfig()` succeeding first, `examples/tutorials/89-readiness-probe` specifically exercises the boundary between "storage is configured but unreachable" (`NOT_READY`, 503) and "storage isn't configured to need reaching at all" (`READY`, memory-backed), which is this exact configuration model in action. `examples/tutorials/113-kms-key-id-resolution` exercises `resolveKmsKeyId()`, one concrete function this configuration ultimately drives (Chapter 3 covers it in depth).

How to validate this yourself

- `packages/shared/src/config/Config.ts`, `ConfigValidation.ts`, `StorageProviders.ts`, `KeyProviders.ts`, `SecretsProviders.ts`, `CryptoAlgorithms.ts`, `TrustProfiles.ts`, the full config model and its validation.
- `packages/api/src/bootstrap/assertStorageConfigured.ts`, `assertSigningKeyMaterialConfigured.ts`, `assertPaytmConnectorConfigured.ts`, the fail-closed startup checks, each with a doc comment explaining exactly what it prevents.
- `packages/api/src/server.ts`, the real, ordered bootstrap sequence.
- `packages/crypto/src/CryptoBootstrap.ts`, `KeyBootstrap.ts`, `SignerBootstrap.ts`, the three composition roots.
- `.env.example`, the canonical, heavily-commented variable reference (verify against `ConfigValidation.ts` for anything you're not sure is still accurate, per the `KEY_PROVIDER` drift noted above).

Integration requirements

A minimal working `.env` needs, at least: `PARMANA_POLICY_DIR`, `PARMANA_KEY_DIR` (with a real `default.private.pem` / `default.public.pem` pair present, or `PARMANA_KEY_MATERIAL_JSON` set), and either `PARMANA_STORAGE=memory` (no further storage config needed) or `PARMANA_STORAGE=supabase` plus `DATABASE_URL` / `SUPABASE_URL` / `SUPABASE_SERVICE_ROLE_KEY` / `SUPABASE_ANON_KEY`. Caller authentication needs either `PARMANA_API_KEYS` populated or `PARMANA_AUTH_DISABLED=true` (development only). `KEY_PROVIDER=aws-kms` additionally needs `AWS_REGION` and either `AWS_ROLE_ARN` (Vercel OIDC federation) or a real AWS credential chain reachable in the runtime environment (`~/.aws/credentials` locally, an instance/task role elsewhere).

Chapter 3: Cryptography

What it is

`packages/crypto` is where every signature Parmana produces or checks actually happens: signing an `ExecutionTrustRecord`, an `ExecutionAuthorization`, a `Receipt`, a `PolicyChangeApprovalRecord`, a caller-audit chain entry, and verifying every one of those independently. It provides a pluggable set of hash and signature algorithms, a deterministic serialization step every one of them depends on, and two key-custody models: local files on disk, and AWS KMS ("sign-without-release," the private key never leaves KMS).

Why it was built

The trust model in Chapter 1 is only as strong as the signatures backing it. Every artifact in the chain of custody claims to be "signed by Parmana," and that claim needs to be checkable by a third party who was never in the room, using nothing but the public key and the artifact itself. That requirement drives three design choices covered below: canonical serialization (so the same logical content always produces the same bytes to sign, regardless of how it was constructed in memory), a `Signer` abstraction separate from `KeyProvider` (so a key that can never be exported, like an AWS KMS key, is a first-class option, not a workaround), and hybrid signing (so a future break of one algorithm doesn't retroactively invalidate everything signed under it, as long as a second algorithm was also applied).

How it works

`CanonicalSerializer`: the thing every signature is actually computed over

`packages/crypto/src/CanonicalSerializer.ts` is nineteen lines of `normalize()` plus a `serialize()` that runs `JSON.stringify` over the result. The normalization rule: objects have their keys recursively sorted lexicographically before serialization; arrays keep their original order (element order is meaningful, key order in an object is not); `Date` becomes its ISO string; everything else passes through unchanged. This single function is what makes two logically-identical objects with different key insertion order, or objects that came from different sources (a hand-typed JSON literal vs. a value round-tripped through a Postgres JSONB column, which does not guarantee to preserve key order) hash and sign identically. Every cryptographic operation in this codebase is required to go through it, `ArtifactSigner.sign()`, `TrustRecordHasher.hash()`, and every `*Crypto` class (`PolicyChangeCrypto`, `RefusalCrypto`, `VerificationCrypto`, `AuditEventCrypto`) all construct their own `CanonicalSerializer` instance rather than hashing raw `JSON.stringify` output directly.

A real, recent incident makes the "array order still matters" half of that rule concrete: on 2026-09-16, a batch of policy approvals initially appeared to fail a content-hash check purely because a database round-trip had reordered object keys in the stored `proposedContent`, that turned out to be a red herring, since `CanonicalSerializer` normalizes object key order away. The real, separate finding underneath it was that the stored content was missing an entire field (`unboundSignalReasons`, and for two policies, `boundSignals`) relative to the current file, a genuine content difference no amount of key-order normalization could paper over. See `docs/CLAIMS.md` §2.26's "Legacy-policy backfill" entry for the full account, worth reading as a caution against assuming a hash mismatch has an obvious cause before actually diffing the two objects field by field.

Hash and signature providers: pluggable, registered, resolved through `CryptoBootstrap`

`packages/crypto/src/providers/hash/SHA256HashProvider.ts` implements `HashProvider`: one method, `hash(data: Uint8Array): Promise<string>`, returning a hex digest via Node's `node:crypto` `createHash("sha256")`. Two more hash algorithms (`sha3-512`, `blake3`) are recognized by `Config.ts`'s validation but have no registered provider class yet, selecting either at the config layer would fail when `HashRegistry.get()` tries to resolve them; `sha256` is the only one with a real implementation.

Signature providers implement `SignatureProvider` (`sign` / `verify`, plus a readonly `algorithm` field): `Ed25519SignatureProvider.ts` (46 lines, wraps `node:crypto`'s `sign` / `verify` with no options object, i.e. pure Ed25519, not Ed25519ph) and `Dilithium3SignatureProvider.ts` (53 lines, near-identical shape, using Node's native `"ml-dsa-65"` key type, this is the ML-DSA-65/FIPS 204 standard, and `ConfigValidation.ts` accepts the string `"ml-dsa-65"` as a config-time alias that resolves to the same internal `"dilithium3"` identifier, so an operator can write either name in

`PRIMARY_SIGNATURE_PROVIDER`). ML-DSA-65 signatures are randomized , signing the same message twice with the same key produces two different, both valid, signatures, noted directly in that provider's own doc comment, worth knowing before assuming a signature mismatch across two runs means something is broken. `MlDsaSupport.ts` detects at runtime (cached after first call) whether the current Node/OpenSSL build actually supports `generateKeyPairSync("ml-dsa-65")`, this requires Node ≥24 with OpenSSL ≥3.5; tests that need it skip cleanly with `ML_DSA_65_SKIP_REASON` on older runtimes rather than failing. Two more algorithms (`ecdsa-p256`, `sphincs-plus`) are recognized by config validation with no registered provider at all.

`CryptoBootstrap.create()` is the actual resolution point: it builds a `CryptoProvider` (hash + signature bundled) by registering the built-in providers into `HashRegistry` / `SignatureRegistry` and selecting by whatever `loadConfig().crypto.hashProvider` / `primarySignatureProvider` say. `createHybrid()` builds both primary and secondary providers at once, throwing if `CRYPTO_MODE=hybrid` but no `SECONDARY_SIGNATURE_PROVIDER` is set. `ProviderFactory` (`providers/ProviderFactory.ts`) is a three-line compatibility wrapper around `CryptoBootstrap.create()`, nothing more.

`Signer` vs. `KeyProvider`: two abstractions for a real constraint

`KeyProvider` (`KeyProvider.ts`) has a `getPrivateKey(keyId): Promise<KeyObject>` method, it assumes the caller can obtain the raw private key material. That assumption is structurally false for AWS KMS, an HSM, or Vault Transit, where the entire point is that the private key never leaves the custody boundary. `Signer` (`Signer.ts`) exists for exactly that case: it drops `getPrivateKey` and adds `sign(keyId, data): Promise<string>` instead, the backend signs on the caller's behalf and only ever returns a signature, never key material. Read operations (`getPublicKey`, `getMetadata`, `hasKey`, `listKeys`) are identical on both interfaces, since verification never needed private key material regardless of custody model.

Two implementations exist for each side today. `FileKeyProvider` (`providers/key/FileKeyProvider.ts`, 192 lines) reads `<keyId>.private.pem` / `<keyId>.public.pem` from `PARMANA_KEY_DIR`, this is the only real `KeyProvider`, and `KeyBootstrap.create()` (Chapter 2) throws for any other `KEY_PROVIDER` value rather than silently falling back to it. `LocalFileSigner` wraps that same file-reading logic behind the `Signer` interface (so `local` custody works through either abstraction), and `KmsSigner` (`providers/signer/KmsSigner.ts`, 217 lines) is the real AWS KMS implementation, resolved only through `SignerBootstrap`, never through `KeyBootstrap`.

`KmsSigner`: sign-without-release, against real AWS KMS

`KmsSigner` supports exactly one key spec/algorithm pair, `ECC_NIST_EDWARDS25519` / `ED25519_SHA_512`, matching this codebase's `ed25519` default, noted in-source as deliberate: AWS KMS added Ed25519 support in November 2025, so adopting it needed no signature-algorithm migration. `sign()` / `getPublicKey()` / `getMetadata()` / `hasKey()` each call a real `KMSClient` command (`SignCommand`, `GetPublicKeyCommand`, `DescribeKeyCommand`); `listKeys()` is deliberately unimplemented, since enumerating every key in an account/region is a broader operation this codebase's routes don't need.

Credentials are never a static access key/secret pair read from this codebase's own configuration. If `AWS_ROLE_ARN` is set, `KmsSigner` dynamically imports the optional `@vercel/oidc-aws-credentials-provider` peer dependency and exchanges Vercel's per-invocation OIDC token for short-lived STS credentials; otherwise it falls back to the AWS SDK's own default credential provider chain.

`resolveKmsKeyId()` is a small, standalone, exported function (`KmsSigner.ts:46-56`) that matters more than its size suggests. Every real signing call site in this codebase passes a *logical* keyId, `"default"`, or a tenant-scoped `"tenant.acme"`, never a raw AWS identifier. AWS KMS's `KeyId` parameter requires a real key ID (UUID), a full ARN, or an alias name/ARN (which must carry the `alias/` prefix); a bare `"default"` matches none of those and AWS rejects it outright. `resolveKmsKeyId()` maps a bare logical keyId to `alias/<keyId>` (mirroring `FileKeyProvider`'s own `<keyId>.private.pem` filename convention), and passes an already-qualified alias, ARN, or raw UUID straight through unchanged. This was a real bug, found by code review before any production traffic hit it: the very first version of this class passed the logical keyId straight through, which meant `KmsSigner.sign("default", data)` would have called AWS with `{ KeyId: "default" }` and failed immediately on every real signing attempt. Because of this function, `alias/default` has to actually exist in AWS, it is not a naming convenience, it is the literal resolution target for the logical id every call site already uses.

`SignerKeyProviderAdapter`: closing a real signing/verification divergence

`providers/SignerKeyProviderAdapter.ts` is the fix for the most serious of the real incidents this migration produced, found against live production traffic on 2026-09-16 (see `docs/VERIFICATION-GAPS.md` G-48/G-49 and the KMS migration troubleshooting guide's item 7). The bug: `EnvelopeVerifier.resolveKey()` uses a `KeyProvider` to look up the public key

for *every* authorization it verifies, including ones signed under the plain `"default"` keyId , not only tenant-scoped ones, contrary to an earlier assessment in this same codebase's own comments that this path was "currently inert." `createExecutionGateway.ts` unconditionally constructed a fresh `new FileKeyProvider()` for that lookup, regardless of `KEY_PROVIDER` . Once `KEY_PROVIDER=aws-kms` was set, every authorization was *signed* by the real KMS key (via `SignerBootstrap`) but *verified* against whatever stale local `default.public.pem` happened to still be materialized from a pre-migration `PARMANA_KEY_MATERIAL_JSON` entry. Signing and verification silently used two different keys, every real request's `signatureVerified` check (and everything that cascades from it: `businessTransactionHashMatches` , `nonceUnseen`) failed. `SignerKeyProviderAdapter` closes this by wrapping the *same* `Signer` instance `createGatewayPublicKey()` already uses, so signing and per-authorization verification now resolve through one identical source, KMS-backed or file-backed, whichever `KEY_PROVIDER` actually says.

`HybridSignatureProvider` : two independent signatures, fail-closed on either

`HybridSignatureProvider.ts` is not a `SignatureProvider` implementer, that interface signs with exactly one key and produces exactly one signature. Hybrid mode needs two of each. `sign()` produces a fixed `[primary, secondary]` pair of `SignatureEntry` values; `verify()` requires *exactly* two entries, one matching each configured algorithm, both independently verified, a missing, extra, duplicated, or mismatched-algorithm entry is rejected outright, never a partial pass. The stated purpose (its own doc comment) is "harvest now, decrypt/forge later" defense for the classical-to-post-quantum transition: if a future quantum computer ever breaks the classical primary algorithm, the post-quantum secondary signature alone still holds, and vice versa if a weakness is ever found in the newer PQ scheme instead. `HYBRID_SIGNATURE_REQUIRED=true` (Chapter 2) makes `VerificationCrypto` reject any record whose `signatures` array is absent or partial, rather than silently falling back to the legacy single-signature check, off by default so turning `CRYPTO_MODE=hybrid` on never retroactively invalidates records signed before that flag was set.

A note on what `.env.example` gets wrong

`.env.example` 's comment on `KEY_PROVIDER` states that only `local` has an implementing class and that setting it to anything else "does nothing." That was true when written and is no longer true: `SignerBootstrap.ts` implements `aws-kms` for real. Separately, a live `.env` in this repository was observed carrying `KMS_REGION` and `KMS_KEY_ALIAS` variables, neither name is read anywhere in `packages/*/src` (confirmed by a repo-wide grep returning zero matches). The variables `KmsSigner` / `assertKmsSigningKeyReachable` actually read are `AWS_REGION` and `AWS_ROLE_ARN` . If you are configuring KMS custody for a real deployment, set those two, not `KMS_REGION` / `KMS_KEY_ALIAS` , the latter pair currently does nothing at all.

How it enables things, with a concrete example

- `examples/tutorials/113-kms-key-id-resolution` exercises `resolveKmsKeyId()` directly against five real input shapes (bare logical id, tenant-scoped id, already-qualified alias, full ARN, raw UUID), no AWS credentials needed since it's a pure string-mapping function.
- `examples/tutorials/114-signing-verification-key-agreement` exercises the `SignerKeyProviderAdapter` fix, proving signing and verification now resolve through the same key source.
- `examples/tutorials/47-canonical-json` demonstrates `CanonicalSerializer` 's normalization behavior directly.
- `packages/crypto/tests/unit/kms-signer.test.ts` and `signer-key-provider-adapter.test.ts` are the permanent, mocked-AWS-SDK regression coverage for both fixes above; the KMS migration's own troubleshooting guide notes a real, one-time end-to-end test was also run against actual AWS KMS before this coverage was trusted, then deleted once confirmed.

How to validate this yourself

- `packages/crypto/src/CanonicalSerializer.ts` , read the whole file, it's short and the normalization rule is the single most load-bearing piece of logic in this chapter.
- `packages/crypto/src/CryptoBootstrap.ts` , `KeyBootstrap.ts` , `SignerBootstrap.ts` , the three composition roots (Chapter 2 also covers these from the config-and-startup angle).
- `packages/crypto/src/Signer.ts` , `KeyProvider.ts` , the two interfaces, both with doc comments explaining exactly why they're separate.
- `packages/crypto/src/providers/signer/KmsSigner.ts` , `providers/SignerKeyProviderAdapter.ts` , the real AWS KMS implementation and the incident it closes.

- `docs/operations/2026-09-15-kms-migration-troubleshooting-guide.md`, the real incident timeline this chapter draws from; each entry names its symptom, root cause, and fix separately, and is worth reading end to end for the migration's own stated lesson: "a migration that changes *where* a system's trust boundary sits needs to be verified against the real target platform's actual runtime behavior, not just a mocked test double of it."
- `docs/VERIFICATION-GAPS.md` G-48/G-49, the ledger entries for the `SignerKeyProviderAdapter` fix specifically.

Integration requirements

For local-file signing (the default): `PARMANA_KEY_DIR` pointing at a directory containing `default.private.pem` / `default.public.pem` (Ed25519 PEM pair), or `PARMANA_KEY_MATERIAL_JSON` set so the process materializes them itself at startup.

For AWS KMS signing: `KEY_PROVIDER=aws-kms`, `AWS_REGION` set, an Ed25519 KMS key (`ECC_NIST_EDWARDS25519` key spec) with an alias matching your logical keyId (`alias/default` for the default signing key), and either `AWS_ROLE_ARN` (for Vercel OIDC federation) or a real AWS credential chain reachable in the runtime environment. `@vercel/oidc-aws-credentials-provider` is an optional peer dependency, only needed if `AWS_ROLE_ARN` is set. Note the Vercel-specific cold-start constraint from the troubleshooting guide's item 3: on Vercel specifically, KMS calls that need the OIDC token cannot run at module top level, they need a real in-flight HTTP request to read the token header from. A long-running process (`server.ts`) is not subject to this, since it calls `assertKmsSigningKeyReachable()` once at real process startup, not per-request.

For hybrid mode: `CRYPTO_MODE=hybrid`, `SECONDARY_SIGNATURE_PROVIDER` set to a different algorithm than `PRIMARY_SIGNATURE_PROVIDER`, and a second key pair present at `<PARMANA_KEY_DIR>/default-secondary.{private,public}.pem`.

Chapter 4: The Policy Engine and Evaluation

What it is

`PolicyEngine` (`packages/policy/src/PolicyEngine.ts`) is the piece of Parmana that turns a policy document and a set of runtime signals into one decision: approve or reject, with a reason and a trace of which rule matched. It is deliberately narrow. Its own doc comment states what it must never do: authorize execution, execute business actions, access external systems, create trust records, replay, or generate timestamps. It is a pure function from `(Policy, PolicySignals)` to `PolicyDecision`.

Why it was built

A policy engine that could reach outside itself (call a database, hit the network, read a clock) would make every decision harder to reason about, replay, or audit. Parmana's design keeps decisioning pure and pushes every side effect (independent verification of signals, persistence, signing) into layers around it. `PolicyEngine` is the load-bearing center that everything else in this chapter's neighboring chapters (signal binding, capability binding, signal-state verification) exists to protect the inputs of.

How it works

The Policy document

A `Policy` (`packages/policy/src/types/Policy.ts:198-300`) has an identity (`policyId`, `policyVersion`, `schemaVersion`), an optional `signalsSchema` describing expected signal types, optional `boundSignals` and `unboundSignalReasons` (covered fully in Chapter 5), and an ordered array of `rules`.

Each `PolicyRule` (`Policy.ts:178-193`) has an `id`, a `condition`, and an `outcome` (`{ action: "approve" | "reject", reason: string }`). Conditions are recursively composable (`Policy.ts:169-173`):

- **Leaf** (`PolicyLeafCondition`): `{ fact, operator, value? }`, a single comparison against one named signal.
- **`all`**: logical AND over child conditions.
- **`any`**: logical OR over child conditions.
- **`always`**: unconditionally true, used as the final catch-all rule.

The full set of supported operators (`PolicyValidator.OPERATORS`, `packages/policy/src/PolicyValidator.ts:32-71`):

Category	Operators
Equality	`eq`, `neq`
Numeric	`gt`, `gte`, `lt`, `lte`, `between`
Collection	`in`, `not_in`, `contains`, `not_contains`, `contains_all`, `contains_any`
String	`starts_with`, `ends_with`, `matches`
Existence	`exists`, `not_exists`
Boolean	`is_true`, `is_false`
Null	`is_null`, `is_not_null`
Length	`length_eq`, `length_gt`, `length_gte`, `length_lt`, `length_lte`
Type	`type_is`

The actual comparison logic lives in `OperatorEvaluator` (`packages/policy/src/OperatorEvaluator.ts`, imported by `PolicyEngine.ts:1`), `PolicyEngine` itself only walks the condition tree and delegates each leaf to it.

Evaluation: first-match-wins

`PolicyEngine.evaluate(policy, signals)` (`PolicyEngine.ts:35-55`) calls a private `findFirstMatch`, which iterates `policy.rules` in array order, pushing each rule's `id` onto a `trace` array as it's visited, and returns the first rule whose condition evaluates `true` against the supplied `signals`. This is deterministic by construction: the same policy and the same signals always produce the same matched rule, because iteration order is fixed array order, not a scored or prioritized search.

A missing fact never satisfies a leaf condition (`PolicyEngine.ts:93-95`: `if (signal === undefined) return false`), an absent signal is not treated as a wildcard or a pass.

If no rule matches, `PolicyDecision.outcome` defaults to `REJECT` via `toOutcome`'s `default` branch (`PolicyEngine.ts:140-142`) with `reason: "no_rule_matched"` and `matchedRuleId: "none"`. Every real policy in this codebase ends with an explicit `always: true` catch-all rule specifically so this default path is never actually reached in practice, it exists as a safety net, not the intended way to reject.

The returned `PolicyDecision` (`packages/policy/src/types/PolicyDecision.ts`) carries `policyId`, `policyVersion`, `outcome`, `reason`, `matchedRuleId`, `evaluatedRules` (how many rules were checked before a match), and `matchedPath` (the full trace of rule IDs visited), enough for an auditor to reconstruct exactly why a decision came out the way it did, without re-running the engine.

Loading and validating: `PolicyRouter`

`PolicyRouter` (`packages/policy/src/PolicyRouter.ts`) is the layer above `PolicyEngine` that actually loads a policy by `(name, version)` via a `PolicyRepository`, then validates it before handing it back. Two validator calls happen on every load:

1. `validator.validate(policy)` (`PolicyRouter.ts:24`), **fail closed**. Throws `PolicyValidationError` on structural problems (missing identity fields, empty `rules`, malformed `boundSignals`/`unboundSignalReasons`, an oversized or dangerous `matches` regex, see below) and, critically, on any rule-referenced fact that is neither in `boundSignals` nor `unboundSignalReasons` (`findUncoveredFacts`, called internally at `PolicyValidator.ts:229`). An uncovered fact is a hard failure, not a warning: every signal a rule can reference must be either bound to the Intent or explicitly acknowledged as independently verified.
2. `validator.findRuleConflicts(policy)` (`PolicyRouter.ts:30`), **advisory only**. Detects rule pairs whose conditions can both be true, which (under first-match-wins) means the earlier rule silently shadows the later one. Logged via `console.warn` with event `policy_rule_conflict_detected`, never thrown. `PolicyValidator.ts:442-466`'s own doc comment explains why: unlike an uncovered fact (unambiguous fix, bind it or acknowledge it), a flagged overlap might be a real bug or the deliberately intended shape (a specific rule followed by a broader fallback). The analysis is also incomplete by design, it only reasons about pairs of single, non-nested-fact conditions; anything involving `all`/`any` is reported as `NEEDS_REVIEW` (level `INFO`) without further analysis.

The `matches` regex guard

Because a `matches` condition is evaluated against live, potentially attacker-influenced signal values, `PolicyValidator.validateRegex` (`PolicyValidator.ts:361-391`) rejects two things before a policy is ever accepted: a pattern longer than 200 characters (`MAX_PATTERN_LENGTH`, `PolicyValidator.ts:331`), and a pattern containing a quantified group whose own contents are themselves quantified (e.g. `(a+)+`), the textbook shape of catastrophic regex backtracking (ReDoS). The doc comment is explicit that this is a heuristic, not a proof: it catches the single-level nested-quantifier case, not every pattern capable of exponential-time backtracking. A linear-time engine or an execution timeout would be needed to close that gap completely; this is a deliberate, bounded improvement over no check at all.

Two `PolicyRepository` implementations

`PolicyRepository` (`packages/policy/src/PolicyRepository.ts`) is a three-method interface: `load`, `save`, `listAll`. Two implementations exist:

- `FilePolicyRepository`, reads/writes `policies/{name}/{version}/policy.json` on the local filesystem. Used in local development and in tests.
- `SupabasePolicyRepository` (`packages/policy/src/SupabasePolicyRepository.ts`), reads/writes a `policies` table via a direct Postgres connection. Added 2026-09-16, the same night as this handbook: Vercel's serverless Functions run on a read-only filesystem, so the very first real production policy approval (`PolicyChangeApprovalService.approve()`'s live-policy write, covered fully in the policy governance chapter) failed with `EROFS` against `FilePolicyRepository`.

`packages/api/src/application.ts` now selects between the two based on whether a real database is configured, constructing whichever one lazily rather than at module import time (see that file's own doc comment for why eager construction was tried first and caused a different bug).

`PolicyRegistry` (`packages/policy/src/PolicyRegistry.ts`) is a separate, much smaller piece, an in-memory `Map` from `"name:version"` to registration metadata (`{ name, version, path }`). Its own doc comment is explicit that it does not load, evaluate, or choose policies; it only tracks what's available.

How it enables things, with a concrete example

- **Tutorial 02** (`examples/tutorials/02-policy-evaluation/run.ts`) exercises `PolicyEngine` directly against a real policy document, showing both the approve and reject paths.
- **Tutorial 04** (`examples/tutorials/04-policy-router/run.ts`) exercises `PolicyRouter` loading a real policy from disk, including its validation step.
- **Tutorial 17** (`examples/tutorials/17-multi-policy-routing/run.ts`) demonstrates routing across multiple distinct policies by name/version.
- **Tutorial 116** (`examples/tutorials/116-supabase-policy-repository/run.ts`, 2026-09-16) demonstrates `SupabasePolicyRepository` directly, including the `EROFS` failure it fixes.

How to validate this yourself

- Read the engine itself: `packages/policy/src/PolicyEngine.ts` (145 lines, worth reading in full).
- Read `packages/policy/src/types/Policy.ts` for the exact shape of every field a policy document can have.
- Run `packages/policy/tests/unit/PolicyEngine.test.ts`, `packages/policy/tests/unit/PolicyValidator.test.ts`, and `packages/policy/tests/unit/PolicyRouter-boundSignals-coverage.test.ts` to see the full assertion suite, including edge cases (missing facts, conflicting rules, oversized regex patterns) not covered above.
- Pick any real policy under `policies/*/*/policy.json` and trace a few signal combinations through its rules by hand, the format is simple enough to do this without running code.

Integration requirements

None beyond what's already required to run the API at all: `PARMANA_POLICY_DIR` (for `FilePolicyRepository`, local/test default `./policies`) or `PARMANA_STORAGE=supabase` plus `DATABASE_URL` (for `SupabasePolicyRepository` in a real deployment). No policy-engine-specific configuration exists, a policy's own content is the only per-policy configuration surface.

Chapter 5: Signal-Intent Binding and Signal-State Verification

What it is

Two separate, deliberately independent checks that run before `PolicyEngine.evaluate` ever sees a request's signals:

1. **Signal-Intent Binding** (`SignalIntentBinder` , `packages/policy/src/SignalIntentBinder.ts`), proves a policy's declared signals describe the *same action* as the Intent that will actually execute if the transaction is approved.
2. **Signal-State Verification** (`SignalStateVerifier` port, `packages/policy/src/types/SignalStateVerifier.ts` , composed via `CompositeSignalStateVerifier`), proves those signals are *true*, not merely consistent.

Both exist because a caller-declared signal, on its own, only ever proves what the caller *says*. Neither check trusts the caller; each closes a different gap in what a bare signal can prove.

Why it was built

`PolicyEngine` evaluates whatever `PolicySignals` it's handed. Nothing about `PolicyEngine` itself prevents a caller from declaring `{"managerApproved": true}` when no manager ever approved anything, or from declaring a small, fully-verified signals payload while the real `Intent` (the thing that actually gets signed and executed) targets something entirely different. `SignalIntentBinder` 's own doc comment states the risk directly (`Policy.ts:238-249`): without it, "a caller could otherwise declare a small, fully-verified signals payload while Intent silently targets something else entirely, and receive a signed APPROVED trust record for it." `SignalStateVerifier` 's doc comment (`SignalStateVerifier.ts:24-33`) draws the companion distinction explicitly: "SignalIntentBinder proves a policy's signals describe the same action as Intent; it never proves those signals are *true*." Two different failure modes, two different, additive mechanisms.

How it works

Signal-Intent Binding

A policy's `boundSignals` field (`Policy.ts:266` , e.g. `{"paymentAmount": "parameters.amount"}`) declares that a given signal key must equal the value found at a specific dot-path into the real `Intent` , an `IntentSnapshot` of `{ target, parameters }` (`SignalIntentBinder.ts:9-12`).

`SignalIntentBinder.findViolations(policy, signals, intent)` (`SignalIntentBinder.ts:67-95`) walks every `boundSignals` entry, resolves the dot-path against the real `Intent` via `resolveIntentPath` (`SignalIntentBinder.ts:32-44` , returns `undefined` for any missing path segment rather than throwing, so a missing Intent field is reported as a mismatch, not a crash), and compares it by strict equality (`!==`) against the caller-declared signal value. Any mismatch becomes a `SignalIntentBindingViolation` , the empty array means either the policy declares no `boundSignals` at all, or every declared binding held.

Only signals with a genuine Intent-side equivalent belong in `boundSignals` , an amount, a target/vendor identifier. A signal representing an external fact with no Intent-side equivalent (`vendorVerified` , `riskScore`) cannot be expressed this way and is deliberately left out, acknowledged instead in `unboundSignalReasons` (Chapter 4 covers the validator's enforcement of this).

Signal-State Verification

The `SignalStateVerifier` interface (`SignalStateVerifier.ts:42-47`) is a single async method: `findViolations(request, signals) => Promise<readonly SignalStateViolation[]>` . A `request` carries `action` , `businessTransactionId` , and `intentParameters` , enough for an implementation to decide whether, and how, to independently re-derive the relevant facts from a real external source and compare them against what the caller declared.

The interface is explicitly optional and capability-scoped: an implementation that doesn't recognize the given `action` returns an empty array, the same "nothing to check" shape `SignalIntentBinder` uses when a policy has no `boundSignals` .

`CompositeSignalStateVerifier` (`packages/policy/src/CompositeSignalStateVerifier.ts`) combines multiple capability-scoped verifiers into one: it queries each in the supplied order and returns the first non-empty result (`CompositeSignalStateVerifier.ts:23-35`). This lets `RuntimeEngine`, which accepts exactly one `SignalStateVerifier`, be wired with as many capability-specific verifiers as production needs, each responsible for only the action(s) it knows how to check.

A concrete production implementation: `createHubSpotSignalStateVerifier` (`packages/api/src/bootstrap/createHubSpotSignalStateVerifier.ts`) builds a `HubSpotSignalStateVerifier` (from `@parmana/connector-hubspot`) wired with the real signing key (`FileKeyProvider`, `DEFAULT_KEY_ID`), the real execution gateway, and an `ApprovalVerifier` for independently checking `preAuthorizedForAmountChange`, a signal that, without this, would be nothing but a caller's unverified claim. Its own doc comment notes `approvalVerifier` is always supplied in production, never omitted; only direct unit tests construct the verifier without it.

Where both run

Both checks run inside the runtime pipeline (Chapter 8 covers the full order) before `PolicyEngine.evaluate`. A violation from either becomes an ordinary policy `REJECT`, no authorization is ever generated for a request whose declared signals don't match the real Intent, or don't match independently verified reality.

How it enables things, with a concrete example

- **Tutorial 62** (`examples/tutorials/62-signal-intent-binding/run.ts`) exercises `SignalIntentBinder` directly, including a deliberate mismatch between declared signals and real Intent parameters.
- **Tutorial 71** (`examples/tutorials/71-hubspot-signal-state-verification/run.ts`) exercises the real `HubSpotSignalStateVerifier` implementation described above.
- **Tutorial 82** (`examples/tutorials/82-composite-signal-state-verification/run.ts`) demonstrates `CompositeSignalStateVerifier` combining multiple capability-scoped verifiers.
- **Tutorial 98** (`examples/tutorials/98-signal-freshness-enforcement/run.ts`) covers a related freshness concern at the execution boundary, worth reading alongside this chapter, though it's a distinct mechanism.

How to validate this yourself

- `packages/policy/src/SignalIntentBinder.ts` (96 lines) and `packages/policy/tests/unit/SignalIntentBinder.test.ts`.
- `packages/policy/src/types/SignalStateVerifier.ts` and `packages/policy/src/CompositeSignalStateVerifier.ts` for the port and its composition.
- `packages/api/src/bootstrap/createHubSpotSignalStateVerifier.ts` for a real, wired production implementation, and its corresponding package under `packages/connector-hubspot/` (or wherever `@parmana/connector-hubspot` actually lives in this checkout) for the verifier's own logic.

Integration requirements

`SignalIntentBinder` needs nothing beyond the policy and the real Intent, both already present in any request. A capability-specific `SignalStateVerifier` (like the HubSpot one) needs whatever real credentials that connector needs to make its own independent verification calls (a HubSpot private app token, in that case), see the connectors chapter for the full credential list per connector.

Chapter 6: Capability and Policy Binding

What it is

`CapabilityPolicyBinder` (`packages/capability-registry/src/CapabilityPolicyBinding.ts`) is a one-method class that enforces a single invariant: for a fixed set of production capabilities (a connector action like `hubspot:deal-update` or `paytm:refund`), the policy the caller declares must be the one canonical policy Parmana designates for that capability, never a different, caller-chosen one.

It lives in its own package, `@parmana/capability-registry` , deliberately depending on nothing except `@parmana/shared` , not on `@parmana/policy` , not on either connector package, so that anything needing to ask "is this capability bound, and to what" can depend on just this map, and `@parmana/policy` itself can depend on it without creating a dependency cycle back through a connector package.

Why it was built

This is a distinct problem from Chapter 5's signal-intent binding, easy to conflate with it because both involve the word "binding." Signal-intent binding proves a policy's declared signals match the real Intent *once a policy has already been selected*. Capability-policy binding is about *which policy gets selected in the first place*.

`CapabilityPolicyBinding.ts` 's own doc comment (lines 74-96) states the gap directly: a capability's `boundSignals` / `SignalStateVerifier` protections are scoped to one specific policy. Nothing upstream enforces that the request is actually evaluated against *that* policy , `PolicyRouter` / `FilePolicyRepository` load whatever `policy.name` / `policy.version` the caller declares, `PolicyEngine.evaluate` takes no `action` parameter at all, and the connector-level capability check only confirms the resolved connector declares the capability, never which policy authorized it. A caller could therefore pair a real capability (`hubspot:deal-update`) with an unrelated, unprotected policy (say, `vendor-payment/2.0.0` , which declares no `boundSignals` for that capability at all), have that unrelated policy's looser rules evaluated against self-declared signals, and get the real `hubspot:deal-update` intent executed anyway , bypassing the capability's intended protections entirely, not merely weakening them.

Found by an independent repository verification (Phase 2K, TD-22) and closed as its own package extraction (G-30 architecture follow-up, see `G-30-ARCHITECTURE-OPTIONS.md` at the repo root and `docs/VERIFICATION-GAPS.md` G-30 for the full history of why this specific package boundary was chosen).

How it works

The canonical binding table

`CANONICAL_CAPABILITY_POLICY_BINDINGS` (`CapabilityPolicyBinding.ts:43-71`) is a `ReadonlyMap` from capability string to `PolicyReference` (`{ name, version, schemaVersion }`). Current real production bindings:

Capability	Bound policy
`hubspot:deal-fetch`	`hubspot-deal-update` @ 1.0.0
`hubspot:deal-update`	`hubspot-deal-update` @ 1.0.0
`github:pr-fetch`	`github-pr-approval` @ 1.0.0
`github:pr-merge`	`github-pr-approval` @ 1.0.0
`paytm:refund`	`customer-refund` @ 1.0.0
`slack:post-message`	`slack-post-message` @ 1.0.0

Every entry corresponds to a capability actually registered in production bootstrap (`packages/api/src/bootstrap/createConnectorRegistry.ts`) and the one policy file purpose-built to authorize it. An action with no entry here, every test/tutorial/example fixture action, and any future capability not yet given a canonical policy, is

entirely unaffected by `CapabilityPolicyBinder`. The table is additive, not a replacement for `PolicyRouter` / `PolicyEngine`; it doesn't change how a selected policy is loaded or evaluated, only whether the caller was allowed to select a different one.

The request-time check

`CapabilityPolicyBinder.findViolation(action, declared)` (`CapabilityPolicyBinding.ts:104-123`) looks up `action` in the canonical table. No entry means nothing to enforce, returns `undefined`. An entry that matches the declared `(name, version)` also returns `undefined`. Any mismatch returns a `CapabilityPolicyBindingViolation` (`{ action, expected, declared }`), which the runtime pipeline (Chapter 8) turns into a rejection before the (wrong) policy file is even evaluated.

The startup-time companion check

A second, separate mechanism enforces the other direction: that every capability a connector actually registers has *either* a canonical binding *or* an explicit, reasoned exemption. `assertConnectorCapabilitiesBound` (`packages/api/src/bootstrap/assertConnectorCapabilitiesBound.ts`) runs once, at startup, from `createConnectorRegistry.ts`, after every connector registration is built and before the registry is handed back to the rest of bootstrap. For each registered capability, it checks `CANONICAL_CAPABILITY_POLICY_BINDINGS` first, then falls back to `INTENTIONALLY_UNBOUND_CAPABILITIES` (`packages/api/src/bootstrap/intentionallyUnboundCapabilities.ts`), a second allowlist for capabilities deliberately left unbound, each entry carrying a reason. A capability found in neither map fails the process at startup, before it can bind a port. The file's own comment notes this assertion currently never fires outside a test-fixture exemption, since every real production capability already has a binding, its purpose is making sure the *next* connector added can't silently ship the same gap `CapabilityPolicyBinder` was built to close.

Together, the two checks form a closed loop: the startup check guarantees every real capability is accounted for (bound or explicitly exempted), and the request-time check guarantees a caller can't route a bound capability through the wrong policy.

How it enables things, with a concrete example

- **Tutorial 83** (`examples/tutorials/83-capability-policy-binding/run.ts`) exercises `CapabilityPolicyBinder` directly, including the exact TD-22 exploit shape (a real capability paired with an unrelated, unprotected policy) and confirms it's now rejected.
- **Tutorial 31** (`examples/tutorials/31-authorization-binding/run.ts`) covers a related but distinct binding concern at the execution-authorization layer, worth reading for contrast, not a substitute for Tutorial 83.

How to validate this yourself

- `packages/capability-registry/src/CapabilityPolicyBinding.ts` (124 lines, read the whole file, every design decision is explained in its own doc comments).
- `packages/capability-registry/tests/unit/CapabilityPolicyBinder.test.ts`, its own header comment restates the exact exploit this class closes, and the test suite proves it.
- `packages/api/src/bootstrap/assertConnectorCapabilitiesBound.ts` and `packages/api/src/bootstrap/intentionallyUnboundCapabilities.ts` for the startup-time companion check.
- `packages/api/src/bootstrap/createConnectorRegistry.ts` to see exactly where and when the startup check runs relative to connector registration.

Integration requirements

None beyond registering a connector at all, `CapabilityPolicyBinder` needs no configuration. Adding a new capability that should be protected this way means adding an entry to `CANONICAL_CAPABILITY_POLICY_BINDINGS`; deliberately leaving one unprotected means adding a reasoned entry to `INTENTIONALLY_UNBOUND_CAPABILITIES` instead, the startup assertion will fail the process if a new capability is registered with neither.

Chapter 7: Policy Governance and the Maker-Checker Flow

What it is

Policy Governance is the mechanism that controls how the content of a `policy.json` file is allowed to change. A policy change cannot take effect the moment someone writes it. It must first be proposed by one authenticated human (the maker), then reviewed and cryptographically approved by a second, genuinely different authenticated human (the checker), before the live policy content updates and a signed, durable approval record is created. The system enforces the maker and checker being different people at the API layer itself, not as a process convention someone could forget to follow.

Why it was built

Before this feature existed, policy authoring sat entirely outside Parmana's own trust model. `GOVERNANCE.md`, `SECURITY.md`, and `TRUST_MODEL.md` all named policy authoring as external to the system's scope. In practice this meant any caller with write access to the `policies/` directory could change what a policy allows or refuses, with no second party involved and no durable, signed record of who made the change or why. Given that policy content decides real outcomes (who gets paid, what gets deployed, what an AI agent is allowed to do), that gap was treated as unacceptable once the system's scope grew to include policy authoring at all. Policy Governance is the answer: a policy change now goes through the same kind of two-person, cryptographically-verified control a bank uses for a wire transfer.

How it works, in full

The lifecycle

A `PendingPolicyChange` (`packages/shared/src/domain/pending-policy-change.ts`) moves through exactly three states, and only ever forward, never back:

```
PENDING_APPROVAL  ->  APPROVED
PENDING_APPROVAL  ->  REJECTED
```

Its fields capture the full proposal: `pendingPolicyChangeId`, the `policyName` and `policyVersion` it targets, the full `proposedContent` (the entire `policy.json`, never a diff or patch), `proposedBy` and `proposedAt`, a required `reason`, and once resolved, `resolvedBy`, `resolvedAt`, and (for a rejection) `rejectionReason`. The type deliberately types `proposedContent` as plain JSON rather than `@parmana/policy`'s `Policy` type, because `@parmana/shared` sits below `@parmana/policy` in the dependency graph and cannot import it without creating an import cycle. Structural validation against the real policy schema happens one layer up, in the API route, before a `PendingPolicyChange` is ever created.

The four endpoints

All four live in `packages/api/src/routes/pending-policy-changes.ts`, mounted at `/policies` in `packages/api/src/app.ts`.

Method	Path	Purpose	Who can call it
`POST`	`/:name/:version/pending-changes`	Propose a change (maker)	Any authenticated human caller (`credentialHolderType: USER`)
`GET`	`/pending-changes?status=...`	List changes, with a `diff` (`current` vs. `proposed`) for each	Any authenticated human caller
`POST`	`/pending-changes/:id/approve`	Approve (checker)	A human caller who is not the proposer, with a valid step-up envelope
`POST`	`/pending-changes/:id/reject`	Reject (checker)	Same as approve, plus a required `rejectionReason`

Every one of the four calls `requireHumanCaller()` first (`packages/api/src/routes/pending-policy-changes.ts:107`). That function checks `isHumanCaller(req.callerCredentialHolderType)` (`packages/api/src/auth/isHumanCaller.ts:22`), which returns true only when the caller's credential was provisioned with `credentialHolderType: AuthorityType.USER`. An undefined value, or `ROLE` / `SERVICE` / `ORGANIZATION`, is denied the same as an unset one; the default is fail closed, never "assume human." A denial is recorded as a signed `caller.non_human_denied` audit event before the request is rejected with `NonHumanCallerDeniedError`. Notably, `isHumanCaller` is called only from these four handlers. It is not wired into the caller-auth middleware globally, so the core execution pipeline (an AI agent submitting a transaction, for example) stays fully actor-agnostic; only policy authorship itself is restricted to verified humans.

Maker cannot equal checker

Both the approve and reject handlers load the existing `PendingPolicyChange` by id and compare `existing.proposedBy === req.callerId` (`packages/api/src/routes/pending-policy-changes.ts:505` and `:605`). A match throws `SameActorCannotApproveOwnChangeError` immediately, before any step-up verification is even attempted. There is no override, no admin bypass, and no configuration flag for this check. It is the literal enforcement of "maker never equals checker."

Step-up authorization (Layer 4)

Passing `requireHumanCaller` and the maker/checker distinctness check is not enough to approve or reject. The request must also carry a `PolicyChangeStepUpAuthorization` envelope, checked by `requireStepUpAuthorization()` (`packages/api/src/routes/pending-policy-changes.ts:165`), which in turn delegates to `PolicyChangeStepUpVerifier` (`packages/api/src/auth/PolicyChangeStepUpVerifier.ts`). This is a second, independent cryptographic proof of intent, on top of (never instead of) the checker's ordinary bearer token.

The envelope is produced entirely offline, on the checker's own machine, by `PolicyChangeStepUpAuthorizationSigner` (`packages/crypto/src/PolicyChangeStepUpAuthorizationCrypto.ts:40`). Signing takes the `pendingPolicyChangeId` and the intended `action` ("approve" or "reject"), stamps a fresh `nonce`, `authorizedAt`, and an `expiresAt` (120 seconds later by default), and signs the whole payload with the checker's Ed25519 step-up private key: a key that is always a separate keypair from the checker's own bearer-token identity, generated independently (`ArtifactSigner` / `Ed25519SignatureProvider`, hardcoded to Ed25519 regardless of whatever signature algorithm the server's own runtime signing key uses, since a step-up key is operator-held and never depends on server-side config at all).

Verification, server-side, runs through `PolicyChangeStepUpVerifier.verify()` (`packages/api/src/auth/PolicyChangeStepUpVerifier.ts:154`), which composes two phases:

1. **`verifyChecks()`**: every side-effect-free check: payload version supported, signature verifies against `req.callerStepUpPublicKey` (the public key on file for the authenticated caller, from `PARMANA_API_KEYS`, never the envelope's own claimed keyId alone), not expired, `pendingPolicyChangeId` and `action` match what this specific request is for, and the envelope's own TTL (`expiresAt - authorizedAt`) does not exceed the server's configured maximum (120 seconds by default, `PolicyChangeStepUpVerifierOptions.maxTtlSeconds`).
2. **`consumeNonce()`**: only called if every check above passed, this burns the envelope's nonce in a dedicated `NonceStore` (`createPolicyChangeStepUpNonceStore.ts`, `SupabasePolicyChangeStepUpNonceStore` in production), so the exact same signed envelope can never be replayed for a second approval. This nonce namespace is entirely separate from execution-authorization or approval-artifact nonces.

A failing check throws `StepUpAuthorizationInvalidError`. The specific reason (missing envelope, expired, replayed, wrong id, wrong action, bad signature, no step-up key provisioned at all) is logged server-side via `console.error` only, and never reaches the HTTP response body: the caller only ever sees a generic 403 with code `STEP_UP_AUTHORIZATION_INVALID`. This is deliberate: the approve/reject endpoint is reachable by any authenticated human other than the proposer, not just the pending change's intended checker, so a granular failure breakdown would hand an attacker holding a stolen bearer token useful reconnaissance (for example, whether a given caller even has a step-up key provisioned) they have no legitimate need for. A real checker debugging their own signing setup has the detail available locally, from the tool that produced the envelope in the first place.

What happens on approve

Once every check passes, the approve handler calls `PolicyChangeApprovalService.approve(existing, req.callerId)` (`packages/api/src/governance/PolicyChangeApprovalService.ts:53`) *before* marking the pending change `APPROVED`. This

ordering is load-bearing: if the service throws partway through, the pending change remains untouched at `PENDING_APPROVAL`, never falsely marked resolved with no corresponding live effect. Inside `approve()`:

1. Loads the current live content at the write target (`proposedContent.policyVersion`, not the pending change's own `policyVersion` field: those can legitimately differ; see `PendingPolicyChange`'s own doc comment on why an in-place patch to an existing version and a version bump are both allowed proposals).
2. Computes `contentHashAfter` (a hash of the proposed content) and, if a prior version existed, `contentHashBefore`.
3. Looks up the most recent `PolicyChangeApprovalRecord` for this same `(policyName, writeVersion)`, if any, and computes `previousRecordHash`: this chains each new record to the one before it, so a deleted or reordered record in the approval history becomes detectable (`verifyPolicyGovernanceIntegrityAtStartup`'s "chain-broken" check, below).
4. Signs the resulting `PolicyChangeApprovalRecord` and persists it via `policyChangeApprovalRecordRepository.create()`: **before** writing the live file.
5. Only then calls `policyRepository.save(policyName, writeVersion, proposedContent)`.

The record-before-file ordering (step 4 before step 5) means the recoverable failure mode is a persisted record with no corresponding file write yet, which a later integrity check can compare against and flag as `"missing"`. The alternative order would risk a live policy silently changing with no durable evidence anything approved it, a governance gap with nothing to point an integrity check at.

Rejections never touch `PolicyChangeApprovalService` at all. A rejection is purely a repository state transition (`resolve()` with `outcome: "rejected"` and the required `rejectionReason`); the rejection reason itself is the durable evidence a rejection leaves behind, since there is no live effect to sign evidence about.

Where the approved content actually lives: two `PolicyRepository` implementations

`PolicyChangeApprovalService.save()` writes through the `PolicyRepository` interface (`packages/policy/src/PolicyRepository.ts`), which has two real implementations:

- **`FilePolicyRepository`** (`packages/policy/src/FilePolicyRepository.ts`): writes `policy.json` to local disk at `<basePath>/<name>/<version>/policy.json`, via a temp-file-then-atomic-rename pattern. Used for local development and tests, where the filesystem is always writable.
- **`SupabasePolicyRepository`** (`packages/policy/src/SupabasePolicyRepository.ts`): added on **2026-09-16**, the same night this book chapter was written. It stores `(policy_name, policy_version) -> content_json` rows in a `policies` table (migration `20260916060000_add_policies_table.sql`) via a direct Postgres connection.

The reason a second implementation exists at all is a real production incident: Vercel's serverless Functions run on a **read-only filesystem**. The very first time `PolicyChangeApprovalService.approve()` ran against a live, deployed instance (all ten of this system's original real policies had sat `PENDING_APPROVAL` for a month before that first real approval), the file write failed immediately with `EROFS`. `application.ts` now chooses between the two implementations based on `config.storage.provider`: memory or test conditions get `FilePolicyRepository`, any real database configuration gets `SupabasePolicyRepository`. See the "What was found and fixed" section below for the full incident chain, including a second bug this same fix introduced and then closed.

Deploy-time and periodic integrity checking

Two further mechanisms exist to catch a governance bypass after the fact: a direct edit to `policies/{name}/{version}/policy.json` that skips the pending-change API entirely:

- **`verifyPolicyGovernanceIntegrityAtStartup()`** (`packages/api/src/governance/verifyPolicyGovernanceIntegrityAtStartup.ts`) compares every approved `(policyName, policyVersion)`'s live content against its most recent approval record's `contentHashAfter`, and separately checks that each record's `previousRecordHash` correctly chains to the record before it. It is deliberately **fail-open**: it never throws, never blocks startup, and every outcome (clean pass, mismatch, or "could not run at all") is logged loudly and distinctly, so "nothing to report" and "couldn't check" never look the same in deploy logs. `schedulePolicyGovernanceIntegrityCheck()` re-runs the same check every 5 minutes for the life of the process, using an `.unref()`'d timer so it can never itself keep the process alive past shutdown.
- **`scripts/verify-policy-changes-approved.ts`** is the opposite discipline: **fail closed**, meant to gate a CI run or a manual pre-deploy check, not to run inside the live server. Given a list of changed policy files (or `--full-scan`), it hashes each file's live content and checks it against `policy_change_approval_records` via a read-only Supabase `anon`

credential, scoped by RLS to that one table. A missing approval record, a content-hash mismatch, and any failure to complete the check at all (Supabase unreachable, a malformed response) are all treated identically as a failure : there is no "couldn't check, let it through" path. `.github/workflows/ci.yml` 's `verify-policy-approvals` job runs it on every push and pull request.

How it enables things, with concrete examples

- **`examples/tutorials/103-policy-governance-maker-checker/run.ts`** : the full manual flow against a real, locally-running Express server: a maker proposes, a `SERVICE` -credentialed caller is denied from proposing at all, the maker is denied from approving its own proposal, a distinct checker with no step-up envelope is denied, and finally a distinct checker with a valid envelope succeeds: proving the live `policy.json` file gets written and a signed approval record gets persisted only once every layer has passed.
- **`examples/tutorials/116-supabase-policy-repository/run.ts`** : proves `SupabasePolicyRepository` satisfies the exact same `PolicyRepository` contract as `FilePolicyRepository` , hermetically, against a minimal fake `pg.Pool` (no real network or database). Also reproduces the real `EROFS` failure directly, against a genuinely read-only temp directory, on platforms where the OS actually enforces it.
- **`examples/tutorials/117-maker-checker-one-shot-scripts/run.ts`** : exercises `scripts/local-review-action.ts` (sign and submit an approve/reject for an existing pending change in one process) and `scripts/refresh-approved-policy-content.ts` (propose, sign, and submit in one process, always from the current on-disk file), and proves both produce the identical durable effects as the fully manual flow. These scripts exist because chaining a hand-signed, 120-second-lived envelope across separate `sign` and `submit` shell commands is genuinely fragile in practice: see the incident notes below.

How to validate this yourself

- The four route handlers and their tests: `packages/api/src/routes/pending-policy-changes.ts` , and `packages/api/tests/integration/pending-policy-changes-governance.integration.test.ts` (23 cases: human-only enforcement on all four endpoints, maker-checker distinctness on approve/reject, every step-up failure mode, file-write-plus-signed-record content, and path-traversal rejection on a malicious `proposedContent.policyVersion`).
- `packages/api/src/governance/PolicyChangeApprovalService.ts` and its unit test `packages/api/tests/unit/PolicyChangeApprovalService.test.ts` , which injects a failure at each step independently and confirms the ordering guarantee described above.
- `packages/api/src/auth/isHumanCaller.ts` and `packages/api/tests/unit/isHumanCaller.test.ts` .
- `packages/api/src/auth/PolicyChangeStepUpVerifier.ts` and `packages/crypto/src/PolicyChangeStepUpAuthorizationCrypto.ts` , plus `packages/api/tests/unit/verifyPolicyGovernanceIntegrityAtStartup.test.ts` (7 cases) for the integrity checker specifically.
- To actually run the real flow against a real deployment, the two operational documents already written for exactly this purpose are `docs/operations/policy-approval-runbook.md` (a copy-pasteable checklist for a specific batch of pending policies) and `docs/operations/policy-governance-developer-guide.md` (the full lifecycle, every command, and a troubleshooting table for every failure mode actually hit while building this feature). This chapter explains the system; those two are the how-to.

Integration requirements

- **Provisioning a checker credential**, one time, per checker:

  ```
  npx tsx scripts/generate-api-key.ts --caller-id "<checker-name>" --credential-holder-type USER --generate-s
  ```

  This prints a bearer key, a step-up **private** key (PEM, never written to disk by the script itself), and an `entry` JSON block containing only the key **hash** and the step-up **public** key: safe to store or transmit, unlike the other two. The operator appends `entry` to the `PARMANA_API_KEYS` environment variable (a JSON array); the checker keeps the bearer key and the private key file on their own machine.
- **The `PARMANA_API_KEYS` entry shape** for a checker, concretely:

```
{
  "callerId": "checker-name",
  "keyHash": "<64-character lowercase hex sha256 of the bearer key>",
  "credentialHolderType": "USER",
  "stepUpPublicKey": "-----BEGIN PUBLIC KEY-----\n...\n-----END PUBLIC KEY-----\n"
}
```

- **`PARMANA_STORAGE`** determines which `PolicyRepository` implementation is live: anything other than `memory` (and `NODE_ENV !== "test"`) selects `SupabasePolicyRepository`, which additionally requires `DATABASE_URL`.
- **`packages/governance-ui`** is an optional, small, server-rendered Express app covering propose, list, and diff-review only. It deliberately has **no** approve or reject routes: its diff page instructs a checker to run `scripts/sign-policy-change-step-up.ts` (or one of the one-shot scripts) locally and submit the result themselves, with their own bearer token, entirely outside the UI. Collecting a step-up private key in a web UI would defeat the one property this whole mechanism exists to guarantee: that the key never leaves the checker's own machine.

What was found and fixed the night of 2026-09-16

This system's ten original real production policies (`access-control`, `connector-capability`, `customer-refund`, `database-change`, `github-pr-approval`, `hubspot-deal-update`, `llm-tool-call`, `production-deployment`, `rag-document-access`, `vendor-payment`) had sat `PENDING_APPROVAL`, proposed on 2026-08-19, untouched for almost a month, because closing that backfill required a genuinely distinct second human checker to become available. The night that checker finally acted, closing it surfaced a real chain of issues, none of them hypothetical:

1. **`EROFS` on the very first live approve.** `PolicyChangeApprovalService.approve()` had never actually executed against a real, deployed instance before that moment. It failed immediately, because `FilePolicyRepository` needs a writable filesystem and Vercel's Functions don't provide one. Fixed by adding `SupabasePolicyRepository` (above).
2. **The first version of that fix introduced a second, real bug.** It constructed `SupabasePolicyRepository(PostgresPoolFactory.create())` eagerly, at `application.ts`'s module scope, rather than lazily. This violated a discipline `packages/api/src/repositories.ts` had already established for every other Supabase-backed repository in this codebase (a `lazyRepository()` Proxy wrapper, closing gap G-15: "importing this module must never itself construct a live Supabase client; only an actual repository call should"). The practical consequence: importing `application.js` at all now opened a real Postgres connection immediately, using whatever `DATABASE_URL` happened to be set at that moment: which broke `examples/tutorials/89-readiness-probe`'s "genuinely unreachable database" test scenario, since the pool singleton (`PostgresPoolFactory` caches one pool process-wide) got created against the *real, working* database before the tutorial ever got a chance to point it at a deliberately unreachable one. Fixed by making `policyRepository` construction lazy, via the same Proxy pattern `repositories.ts` already uses.
3. **`PostgresPoolFactory.create()` had no `connectionTimeoutMillis`.** Found via the same tutorial: connecting to a genuinely unreachable address hung indefinitely instead of failing fast. Fixed with a 5-second timeout.
4. **The original 2026-08-19 proposals had gone stale.** By 2026-09-16, all ten live policy files had gained an `unboundSignalReasons` documentation field the original proposals never had (harmless, non-functional), and two of them (`connector-capability`, `customer-refund`) had gained a `boundSignals` mapping the original proposals were also missing entirely: a real, functional gap, since `boundSignals` is what lets the runtime derive a signal like `paymentAmount` directly from the request's own parameters rather than requiring independent verification. Approving the stale content as-is would have silently persisted that gap into the newly-created `policies` table. Instead, all ten (plus four more pre-existing policies that had never been proposed at all) were re-proposed with their current, correct file content and re-approved, confirmed clean via `scripts/verify-policy-changes-approved.ts --full-scan`.

The full, first-hand account of this incident chain, including exact timestamps and every resulting `policyChangeApprovalRecordId`, is recorded in `docs/CLAIMS.md` §2.26's "Legacy-policy backfill" entry.

What remains genuinely open

`POLICY_EXECUTION_VERIFICATION_ENFORCED` (see Chapter 5's sibling material on execution-time verification, and `docs/CLAIMS.md` §2.35) remains `false` by design. This flag, when enabled, makes `RuntimeEngine` refuse execution against any policy with no approval record, an invalid approval-record signature, or content that no longer matches its approval record. Now that all fourteen real production policies have a genuine approval record, the original reason the flag defaulted off (an unconditional gate would have refused every execution in the system) no longer holds. Turning it on is nonetheless

a **separate, deliberate decision**, explicitly reserved for a dedicated step in `docs/operations/policy-approval-runbook.md` Part 5, not something that happens as a side effect of closing the backfill. As of this writing, it has not been turned on.

Chapter 8: The Runtime Pipeline

What it is

The Runtime Pipeline is the ordered sequence of checks and steps a `BusinessTransaction` passes through, from arrival to a signed `ExecutionTrustRecord`. Its center is `RuntimeEngine.execute()` (`packages/runtime/src/RuntimeEngine.ts`), a single method that loads a policy, runs every configured pre-authorization protection in a fixed order, evaluates the policy, builds a `Decision`, signs an `ExecutionAuthorizationPayload` if approved, and hands the result to two further pipelines (`RuntimePipeline` for execution, `BusinessTrustPipeline` for the final trust record). Everything downstream of a request being accepted runs through this one method.

Why it was built

A system that authorizes real-world actions needs one place where every protection is guaranteed to run, in a guaranteed order, with no way for a caller or a misconfigured deployment to skip a step silently. `RuntimeEngine` is that place. Its own class doc comment states its responsibilities directly: load the requested policy, evaluate it deterministically, create the `Decision` artifact, create the initial `Execution` artifact, execute the Runtime Pipeline, and produce the `ExecutionTrustRecord`. Most of the protections wired into it (signal-intent binding, capability/policy binding, signal-state verification, policy governance verification) were added incrementally, each as an **optional, trailing constructor parameter**, specifically so every pre-existing call site keeps compiling and behaving identically when a new protection is introduced. This is a deliberate, repeated pattern in this codebase, not an accident of growth.

How it works

`RuntimeEngine`'s constructor takes ten required dependencies (`RuntimePipeline`, `PolicyRouter`, `PolicyEngine`, `SignalIntentBinder`, `DecisionBuilder`, `ExecutionGate`, `ExecutionBuilder`, `BusinessTrustPipeline`, `RuntimeAuthorizationSigner`, `authorizationTtlSeconds`) and seven optional trailing ones (`hooks`, `refusalRecordBuilder`, `refusalRecordRepository`, `signalStateVerifier`, `capabilityPolicyBinder`, `policyExecutionVerifier`, `policyGovernanceAnchorResolver`). At construction, it logs which optional protections are actually configured (`runtime_engine_constructed`), which is how an operator can confirm, from log output alone, exactly which protections are active for a given deployment.

`execute(transaction)` runs these steps, in this exact order (comments quoted verbatim from `RuntimeEngine.ts`):

1. **Policy load.** `policyRouter.load(transaction.policy.name, transaction.policy.version)`. `beforePolicyLoad` / `afterPolicyLoad` hooks fire around this.
2. **Policy content hash (G-24).** `policyContentHash = policyContentHasher.hash(policy)`, computed from the *actually loaded* policy document, never from what the caller declared, using the same `TrustRecordHasher` (canonicalize, then SHA-256) every other content hash in this codebase uses.
3. **Policy Governance evidence anchor (G-45).** If `policyGovernanceAnchorResolver` is configured, it resolves whether this policy has a valid, matching approval record. Purely evidentiary, a resolver error is caught and logged, never allowed to affect the real outcome.
4. **Policy Governance execution-time verification.** If `policyExecutionVerifier` is configured, it checks the same approval-record question, but this one *can* reject: "a policy with no approval record, an approval record whose signature does not verify, or live content that no longer matches its approval record is rejected before `PolicyEngine` ever evaluates a single rule in it."
5. **Capability/Policy binding (TD-22).** If `capabilityPolicyBinder` is configured and step 4 found no violation, it checks whether the invoked capability has a canonical policy binding and, if so, whether the declared policy matches it. Runs before signal-intent binding "for the same reason capabilityPolicyBinder runs before signalIntentBinder: checking a narrower guarantee against an already-wrong policy is meaningless."
6. **Signal-intent binding.** If steps 4 and 5 found no violation, `signalIntentBinder.findViolations(policy, signals, {target, parameters})` checks that every `boundSignals` entry actually matches the real Intent's target/parameters.
7. **Policy evaluation.** If none of steps 4 through 6 rejected, `policyEngine.evaluate(policy, signals)` runs the real rules. Any rejection from steps 4 through 6 becomes a synthetic `PolicyDecision` with `outcome: REJECT`, `evaluatedRules: 0`, no rule is ever evaluated for a request that fails an earlier check.

8. **Signal-state verification (G-24 residual closure, RFC-0022).** Only runs if the provisional decision is `APPROVE`, "a request already rejected... needs no independent re-fetch of real state." If `signalStateVerifier` is configured, it independently re-derives the declared facts from a real external source; a mismatch overrides the decision to `REJECT`.
9. **Decision.** `decisionBuilder.build(transaction, policyDecision)`.
10. **Refusal Record (RFC-0021).** If the decision is not `APPROVED`, a `RefusalRecord` is built and persisted, but this write is explicitly never allowed to affect or block anything downstream; a failure here is only logged (`refusal_record_write_failed`), the method quote: "The refusal itself must never depend on its own evidence being writable."
11. **Enforce.** `executionGate.enforce(decision)`, this is the actual gate; a `REJECT` throws here and nothing past this point runs for a rejected transaction.
12. **Authorization.** Only reached on `APPROVE`. `signalsHash` is computed, then `authorizationSigner.sign(...)` produces the `SignedExecutionAuthorization` (Chapter 9 covers this envelope's exact fields).
13. **Execution + Runtime Context.** `executionBuilder.build(...)` and the `RuntimeContext` are assembled, the context carries a *copy* of `transaction.policy` augmented with `contentHash` and, if resolved, `governanceAnchor`; the original caller-submitted transaction (already persisted before this method ever ran) is never mutated.
14. **Runtime Pipeline, then Business Trust Pipeline.** `pipeline.execute(context)` runs the actual execution stages (Chapter 10 covers `ExecutionGateway`, one implementation of the `ExecutionSystem` interface this pipeline calls into); `trustPipeline.execute(...)` produces the final, signed `ExecutionTrustRecord`.

`RuntimeFactory.create()` (`packages/runtime/src/RuntimeFactory.ts`) is the composition root that assembles a fully wired `RuntimeEngine` (via `RuntimeBuilder`) plus the surrounding `ExecutionTrustApplication` (transaction/execution/verification/receipt services). It takes the same optional protections as trailing parameters and only wires them into `RuntimeBuilder` when supplied (`if (signalStateVerifier) { builder.withSignalStateVerifier(...) }`), the same "absent means unconfigured, not broken" discipline as the constructor itself.

How it enables things, with examples

- `examples/tutorials/03-runtime-execution`, the baseline: a transaction through `RuntimeEngine.execute()` to a trust record, no optional protections.
- `examples/tutorials/16-runtime-pipeline`, the pipeline stages themselves.
- `examples/tutorials/18-runtime-hooks`, the `RuntimeHook` interface (`beforePolicyLoad`, `afterDecision`, etc.) that lets an integrator observe or extend the pipeline without modifying `RuntimeEngine` itself.
- `examples/tutorials/19-runtime-composition`, composing multiple pipeline stages.
- `examples/tutorials/15-custom-runtime-component`, writing a custom pipeline stage.

How to validate this yourself

- `packages/runtime/src/RuntimeEngine.ts`, the method itself; every ordering claim above is a direct comment in this file.
- `packages/runtime/src/RuntimeFactory.ts`, `RuntimeBuilder.ts`, how a `RuntimeEngine` actually gets constructed for a real deployment.
- `packages/runtime/tests/e2e/runtime.e2e.test.ts`, end-to-end proof, including the G-24 content-hash-at-decision-time assertion against real on-disk policy content.
- `packages/runtime/tests/unit/optional-protections-logging.test.ts`, proves the construction-time log line accurately reflects what's wired.
- `packages/runtime/tests/integration/runtime.integration.test.ts`, the fuller integration surface.

Integration requirements

None beyond what Chapter 2 (Configuration and Bootstrapping) already covers, `RuntimeEngine` itself takes no environment variables directly; every dependency it needs is constructed and passed in by `RuntimeFactory` / `application.ts`. The optional protections (`signalStateVerifier`, `capabilityPolicyBinder`, `policyExecutionVerifier`, `policyGovernanceAnchorResolver`) each have their own configuration surface, covered in the chapters specific to them.

Chapter 9: The Execution Authorization Envelope

What it is

A `SignedExecutionAuthorization` is the cryptographic proof that Parmana decided a specific request should execute. It's what crosses the boundary between "Parmana decided APPROVE" and "a connector actually did something in the real world." Its own doc comment states the guarantee directly: "Enterprise systems should execute only requests carrying a valid, verified `SignedExecutionAuthorization`." Nothing downstream is trusted to re-derive that a decision was made; the envelope carries the proof itself.

Why it was built

Without a signed, independently verifiable authorization, a connector (or anything holding the ability to call one) would have to trust that whatever called it had genuinely gone through `RuntimeEngine`, with no way to check. The envelope makes "was this actually authorized, by this exact decision, under this exact policy content, for this exact content, within this time window" a question anyone downstream can answer themselves, from the envelope alone, without a network call back to Parmana. Every field on the payload (`packages/shared/src/domain/execution-authorization.ts`) exists to close one specific gap this system's own incident history (Chapter 22) found real.

How it works

`ExecutionAuthorizationPayload` (the signed content) carries:

Field	Purpose
`version`	Must be `1`. A verifier rejects any other value, including a missing field, before even attempting signature verification.
`authorizationId`	Unique identifier for this authorization.
`nonce`	Single-use. A receiving system must reject a previously-seen nonce, replay protection.
`decisionId`	The approved `Decision` this authorization corresponds to.
`businessTransactionId`	The transaction.
`policyName` / `policyVersion`	Which policy produced the decision.
`authorizedAt` / `expiresAt`	ISO-8601 UTC. A receiving system must reject an expired authorization.
`businessTransactionHash`	Canonical hash of the `ExecutableContent` (`businessTransactionId`, `action`, `target`, `parameters`) approved for execution, computed identically on the signing side (from the runtime's in-memory transaction) and the verifying side (from the JSON-parsed request), so a mismatch means the payload was modified after signing while keeping the same `businessTransactionId`.
`policyContentHash` (optional)	The G-24 content hash of the *exact policy document* that produced this decision, not merely the version string. Optional so authorizations signed before this field existed keep verifying, absent means "not covered by the policy-freshness check," never "policy is stale."
`signalsHash` (optional)	Canonical hash of the runtime signals evaluated for this decision, the real-world conditions (vendor status, risk exposure, etc.) that justified it, distinct from both the content hash and the policy hash.
`submittedBy` (optional)	The authenticated caller identity, when caller-auth was enabled.
`grantedCapability` (optional)	The capability the caller-to-capability check confirmed this caller was permitted to invoke, carried forward as a signed fact rather than discarded once the request passes, a receiving system can confirm this equals the executed action, catching any divergence between what a caller was cleared for and what's actually presented for execution.

The complete envelope (`SignedExecutionAuthorization`) wraps this payload with `signature` (over the canonical serialization of the payload, see Chapter 3 for `CanonicalSerializer`), `keyId` (so a verifier can select the right public key),

and `algorithm`.

Signing happens in `RuntimeAuthorizationSigner` (`packages/runtime/src/RuntimeAuthorizationSigner.ts`), which composes `@parmana/crypto`'s `AuthorizationSigner` against whichever `Signer` (`LocalFileSigner` or `KmsSigner`, resolved once through `SignerBootstrap`, see Chapter 3) is configured, using a per-tenant key resolved through `TenantKeyResolver` (falling back to the shared `"default"` key when no tenant-specific key exists). `RuntimeEngine.execute()` calls this once, at the "Authorization" step (Chapter 8, step 12), only ever reached after policy evaluation approved the request.

Verification happens twice, in two different places, checking different things:

1. **`@parmana/envelope-verifier`'s `EnvelopeVerifier`**, used by `ExecutionGateway` (Chapter 10), checks the envelope itself: version, signature, expiry, TTL policy, then the `businessTransactionHash` recompute-and-compare, then (if wired) the policy-freshness and signal-freshness checks, then nonce consumption last (the only side-effecting check, run only once everything else has passed).
2. Anyone holding the receiving system's copy of Parmana's public key can independently re-verify the same signature offline, without calling back to Parmana at all, this is what Chapter 18 (Independent Verification) covers.

How it enables things, with examples

- `examples/tutorials/11-execution-authorization`, the payload itself, signed and inspected.
- `examples/tutorials/26-execution-authorization-verification`, independent verification.
- `examples/tutorials/27-authorization-expiration`, the `expiresAt` check rejecting a stale envelope.
- `examples/tutorials/29-authorization-tampering`, mutating a signed payload and watching signature verification catch it.
- `examples/tutorials/31-authorization-binding`, the `businessTransactionHash` binding content to the authorization.
- `examples/tutorials/41-expired-authorization`, `43-stolen-authorization`, further negative cases against the same envelope.

How to validate this yourself

- `packages/shared/src/domain/execution-authorization.ts`, the type definitions; every claim above about a field's purpose is quoted or closely paraphrased from this file's own doc comments.
- `packages/runtime/src/RuntimeAuthorizationSigner.ts`, the signing side.
- `packages/crypto/src/AuthorizationSigner.ts`, `AuthorizationVerifier.ts`, the shared signing/verification logic.
- `packages/envelope-verifier/src/`, `EnvelopeVerifier`'s own check ordering.
- `packages/runtime/tests/unit/execution-authorization-wiring.test.ts`, proves the fields get populated correctly from a real `RuntimeEngine.execute()` call.

Integration requirements

- A configured signer (`KEY_PROVIDER=local` with `PARMANA_KEY_DIR`, or `KEY_PROVIDER=aws-kms` with the relevant AWS configuration, see Chapter 3).
- `EXECUTION_AUTHORIZATION_TTL_SECONDS` (defaults to 120 per this repo's own `.env`) controls how long a signed authorization remains valid before a receiving system must reject it.
- A receiving system that wants to verify authorizations itself needs Parmana's public key, see Chapter 18 for the discovery mechanism (`GET /keys/:keyId`, `/.well-known/jwks.json`).

Chapter 10: The Execution Gateway

A correction, first

An older architecture document in this repository assumed `ExecutionGateway` lives at `packages/api/src/execution-gateway/ExecutionGateway.ts`. That path does not exist. The real, current location is `packages/execution-gateway/src/ExecutionGateway.ts`, a separate, top-level package, not something nested inside `packages/api`. This chapter is written directly from that real file.

What it is

`ExecutionGateway` implements `@parmana/execution-system`'s `ExecutionSystem` interface, the same seam `RuntimeFactory.create()` (Chapter 8) accepts any implementation of. Its own class doc comment calls it "the sole release boundary" for whichever `Connector` it's constructed with: no execution reaches a connector without passing through this gateway's full verification sequence first.

Why it was built

A signed `ExecutionAuthorizationPayload` (Chapter 9) proves a decision was made, but something still has to actually check that proof before letting a connector run, and that something needs to independently re-verify, not just trust, every claim the envelope makes, including claims about the world having stayed the same since the envelope was signed (the policy hasn't changed, the underlying signals haven't drifted). `ExecutionGateway` composes `@parmana/envelope-verifier`'s `EnvelopeVerifier` (signature/expiry/TTL/nonce) and adds two further checks of its own: a content-hash recompute-and-compare, and, when wired, policy-freshness and signal-freshness checks against the *current* state of the world, not just the state at authorization time.

How it works

Construction (`ExecutionGatewayOptions`) requires a `publicKey` (or a `keyProvider` for keyId-aware, rotation-safe lookup) and a `nonceStore`, and exactly one of a `connector` or an `executionControl` (constructing with both, or neither, throws immediately). Optional `policyRepository` and `signalStateVerifier` enable the two additive checks below; omitting either skips that specific check rather than failing closed on it.

`verify(request, now)` runs the checks in this exact order, quoted directly from the class doc comment, "Session 3's ordering rule: side-effect-free checks first, nonce consumed last and only on success":

1. `version -> signature -> expiry -> TTL policy` (all inside `EnvelopeVerifier.verifyChecks()`).
2. `businessTransactionHash` recompute-and-compare, only attempted if step 1 passed.
3. `policyStillCurrent` recompute-and-compare (when a `PolicyRepository` is wired and the authorization carries a `policyContentHash`), recomputes the *current* content hash of the named policy and compares it to what the authorization was signed under. If the policy no longer exists at all (e.g. replaced in place by a governed change), this is treated identically to a content mismatch, not a separate error case.
4. `signalsStillCurrent` recompute-and-verify (when a `SignalStateVerifier` is wired and both the request and authorization carry signals/`signalsHash`), first a hash comparison (catching a request whose signals were altered after authorization), then an independent re-verification of those signals against real-world state via the verifier.
5. **Nonce consumption, last, only if every prior check passed.** This is the only side-effecting check in the sequence, "a mismatched or forged request must not burn a nonce."

`execute(request)` calls `verify()`, and on any failure throws, specifically, `NonceAlreadyConsumedError` when nonce replay is the *sole* failing check (every other check passed), or a generic `Error` naming every failing check and both hash values on a content mismatch otherwise. `isSoleFailureNonceReplay()` is the precise logic for that distinction: strip out `nonceUnseen`, and check every remaining value is `true` or `undefined` (an undefined check means "skipped," which still counts as passing for this purpose). On success, the verified `executableContent` is deep-frozen and handed to either the legacy `Connector` interface (`this.connector.execute(...)`) or the newer `executionControl` path (a `service.execute(...)` call with a freshly

minted or static `gatewayAuthentication` value, or an `ExecutionChannel.release(...)` call for the deprecated channel-based path).

How it enables things, with examples

- `examples/tutorials/34-execution-gateway`, the gateway directly.
- `examples/tutorials/33-execution-boundary`, the boundary concept: what does and doesn't cross it.
- `examples/tutorials/32-execution-pipeline`, the gateway inside the larger pipeline.
- `examples/tutorials/81-connector-execution-gateway`, a real connector behind the gateway.
- `examples/tutorials/86-gateway-attestation`, attestation semantics at this boundary.

How to validate this yourself

- `packages/execution-gateway/src/ExecutionGateway.ts`, the full verification sequence; every ordering and failure-handling claim above is a direct comment or method in this file.
- `packages/execution-gateway/src/GatewayVerificationResult.ts`, the structured result shape (`checks`, `hashMismatch`, `policyContentMismatch`, `signalsHashMismatch`, `signalDivergence`).
- `packages/execution-gateway/tests/unit/execution-gateway.test.ts`, the primary unit coverage.
- `packages/execution-gateway/tests/unit/execution-gateway.dilithium3.test.ts`, the same gateway logic under a post-quantum signature algorithm (see Chapter 3).
- `packages/execution-gateway/tests/unit/credential-non-exposure.test.ts`, a separate but closely related guarantee this package also carries, covered in full in Chapter 11 (Credential Isolation).

Integration requirements

- A public key (or `KeyProvider`) matching whatever signed the authorizations this gateway will verify.
- A `NonceStore` implementation, `MemoryNonceStore` for tests/tutorials, `SupabaseNonceStore` (via the storage layer, Chapter 13) for a real deployment, so replay protection survives a process restart.
- To enable policy-freshness checking: a `PolicyRepository` (Chapter 4/13) pointed at the same policy content the signing side used.
- To enable signal-freshness checking: a `SignalStateVerifier` (Chapter 5) capable of re-deriving the relevant real-world facts.
- Exactly one of a `Connector` implementation or an `executionControl` configuration , Chapter 12 covers what a real `Connector` looks like.

Chapter 11: Credential Isolation

What it is

Credential isolation is the guarantee that a connector, the code that actually calls an external service like HubSpot, GitHub, or Paytm, never holds a long-lived, reusable credential. Instead it receives a single-use, time-bounded session credential, minted fresh for one specific execution, resolved to the real secret only at the last possible moment, and destroyed immediately after use, whether that use succeeded or failed.

Why it was built

A connector is the part of this system closest to the outside world, and therefore the part most exposed to a bug, a dependency vulnerability, or a compromised third-party API response. If a connector held the platform's real HubSpot API key or GitHub App private key directly, a single flaw in connector code could leak that credential, and the leak would be reusable by whoever obtained it, for as long as the credential remained valid. Credential isolation removes that exposure at the architecture level: even a fully compromised connector can only ever hand an attacker a credential that is already spent, or one that is about to expire in seconds and is bound to an authorization that has already been consumed.

How it works

The mechanism lives in `packages/execution-control/src/`, specifically two classes:

`SessionCredentialVault` (`SessionCredentialVault.ts:18-25`) exposes three operations: `issue()`, `consume()`, and `revoke()`. Its in-memory implementation, `InMemorySessionCredentialVault`, wraps an underlying `CredentialVault` (the thing that actually holds the real, long-lived secret) and never itself stores the secret value. The class's own doc comment states this precisely: "issue() only confirms the underlying credential exists, it never resolves or stores the secret" (`SessionCredentialVault.ts:44-46`).

- `issue(connectorId, authorizationId)` calls `credentials.getCredential(connectorId)` once, purely to confirm a credential exists for that connector, then discards the resolved value immediately. It returns a `SessionCredential` object (`SessionCredentialVault.ts:10-16`) containing only an id, timestamps, and the connector/authorization it's scoped to, never the secret itself.
- `consume(sessionCredentialId)` is the only method that ever calls back into the underlying vault for the real value, and it does so fresh, at the moment of use. Before it does, it checks the session record for three failure conditions, each throwing if true: already revoked, already used, or expired (`SessionCredentialVault.ts:97-111`). It then marks the session `used = true` and returns the real credential exactly once. A second call to `consume()` on the same session id always throws.
- `revoke(sessionCredentialId)` marks a session dead, unconditionally.

`SessionCredentialSecureConnector` (`SessionCredentialSecureConnector.ts`) is what actually wraps a real connector's `execute()` call with this lifecycle. Its own `execute()` method (`SessionCredentialSecureConnector.ts:74-131`) does exactly this, in order:

1. Asserts the connector's policy allows this specific request (`policy.assertAllowed(...)`).
2. Calls `sessionCredentials.issue(...)`, obtaining a session credential id.
3. Inside a `try` / `finally`: calls `sessionCredentials.consume(...)` to resolve the real credential just before use, passes it straight into the wrapped executor's `execute()` call, and in the `finally` block, unconditionally calls `sessionCredentials.revoke(...)`, this runs whether the executor succeeded, threw, or the policy check itself threw after issuance.
4. Records exactly one audit event, `execution.completed` or `execution.rejected`, naming the connector, the session credential's id (never its value), the authorization it executed under, and the capability invoked. The credential's id is auditable; the credential's value never appears anywhere in the audit trail.

Because `consume()` throws on a second call, and `revoke()` runs unconditionally on every exit path via `finally`, a session credential can be used at most once, ever, no matter how the execution ends.

One subtlety worth being precise about, from the class's own doc comment (`SessionCredentialSecureConnector.ts:20-32`): this connector is also constructed with a `gatewayAuthentication` value proving it was configured with material signed by the Gateway's key at registration time. That value is frozen once, at construction, and reused for every execution this connector instance will ever handle, it is not, and cannot be, a per-request proof by itself. The actual per-request replay protection comes from a different layer, `ConnectorPolicy` 's single-use `GatewaySession` check, enforced one level up, before a `GatewaySession` is even created. The two guarantees (registration-time authenticity, and per-request single-use) are deliberately separate mechanisms, not the same check reused twice.

How it enables things, with a concrete example

`packages/api/tests/integration/credential-isolation.integration.test.ts` proves this guarantee at the HTTP boundary, not just at the library level. Its own doc comment (`credential-isolation.integration.test.ts:10-19`) explains the split precisely: `packages/execution-control/tests/unit/session-credential-secure-connector.test.ts` already proves "a session credential is issued and destroyed exactly once per execution" by calling `SessionCredentialSecureConnector.execute()` directly; this integration test proves the exact same guarantee holds through a real `POST /execute` HTTP request against the real Express app, through the real `ExecutionGateway` / `SessionCredentialExecutionControl` / `ExecutionControlService` / `SessionCredentialSecureConnector` chain. It builds its own "inspectable" execution system (`packages/api/tests/bootstrap/createInspectableExecutionSystem.ts`) specifically to get a direct reference to the audit sink and the session credential vault, which the production bootstrap chain does not expose by design. The test asserts that after a successful `/execute` call, exactly one `execution.completed` audit event carries a `credentialId` , confirming a session credential really was issued and consumed, not skipped.

`packages/api/tests/integration/execution-failure.integration.test.ts` is the companion case: it proves the same revoke-on-every-exit-path guarantee when the underlying connector executor throws, not just on the success path.

How to validate this yourself

- `packages/execution-control/src/SessionCredentialVault.ts` and `SessionCredentialSecureConnector.ts` , the actual mechanism.
- `packages/execution-control/src/types.ts` , `CredentialVault` , `ExecutionCredential` , `SecureConnector` , `ConnectorPolicy` interface shapes.
- `packages/execution-control/tests/unit/session-credential-vault.test.ts` , `session-credential-secure-connector.test.ts` , `session-credential-execution-control.test.ts` , the library-level proofs, one per class.
- `packages/api/tests/integration/credential-isolation.integration.test.ts` and `execution-failure.integration.test.ts` , the HTTP-boundary proofs.
- `packages/api/tests/bootstrap/createInspectableExecutionSystem.ts` , how a test gets a direct handle on the audit sink and vault that production code never exposes.

Integration requirements

None from an operator's perspective, this is an internal architectural guarantee, not a configurable feature. A new connector wired into this system automatically gets session credential isolation by virtue of being registered behind `SessionCredentialSecureConnector` ; there is no opt-out path documented in the source, and none should be added without a strong, explicit reason.

Chapter 12: Connectors

What it is

A connector is the piece of code that actually performs a real-world action, a HubSpot deal update, a GitHub pull request merge, a Paytm refund, a Slack message, after every governance check upstream (policy evaluation, signal verification, capability binding, execution authorization) has already approved it. Connectors are deliberately dumb: they validate requests, execute one already-authorized operation using an already-resolved credential (Chapter 11), and return a deterministic response. They never evaluate policy, never authorize anything, and never resolve credentials themselves.

Why it was built

Every governed action this system can take needs an actual integration with the outside world, and that integration needs to be swappable, testable in isolation, and structurally incapable of bypassing the governance layers above it. The `Connector` contract (`packages/connector-sdk/src/ConnectorTypes.ts:104-111`) exists to enforce that boundary in code, not just in documentation: its own doc comment states plainly that "a Connector never receives a request directly from the Runtime or AI, only from `SdkConnectorExecutor` , itself only reachable through execution-control's `SecureConnector` , itself only reachable through the Execution Gateway" (`ConnectorTypes.ts:99-102`).

How it works

The `Connector` interface itself is small:

```
export interface Connector {
  readonly connectorId: string;
  readonly capabilities: ConnectorCapabilities;
  execute(
    request: ConnectorRequest,
    context: ConnectorExecutionContext,
  ): Promise<ConnectorResponse>;
}
```

`ConnectorExecutionContext` (`ConnectorTypes.ts:85-89`) carries a `credential` (a `CredentialHandle` , already resolved by the session credential vault from Chapter 11, not a raw secret the connector has to go fetch), a `timeoutMs` , and a `requestedAt` timestamp.

Each real connector in this codebase splits into two files: a `*-Capabilities.ts` file in a dedicated `@parmana/connector-{name}` package (pure metadata: capability identifier constants and request/response DTOs, no execution logic at all), and an executable adapter under `packages/execution-gateway/src/connector-execution/Gateway{Name}Adapter.ts` that actually performs the HTTP call. This split is deliberate and consistent across every connector; each capabilities file says so explicitly in its own header comment (e.g. `HubSpotCapabilities.ts:1-5` , `GitHubCapabilities.ts:1-5`).

The four real, live connectors

Connector	Capability string(s)	Governing policy	Status
HubSpot	`hubspot:deal-fetch` , `hubspot:deal-update`	`hubspot-deal-update@1.0.0`	Live
GitHub	`github:pr-fetch` , `github:pr-merge`	`github-pr-approval@1.0.0`	Live
Paytm	`paytm:refund`	`customer-refund@1.0.0`	Live
Slack	`slack:post-message`	`slack-post-message@1.0.0`	Live
Razorpay	(historical)	(historical)	Removed 2026-08-12

This table's policy column is read directly from `packages/capability-registry/src/CapabilityPolicyBinding.ts:43-71` , `CANONICAL_CAPABILITY_POLICY_BINDINGS` , the same table Chapter 6 covers in depth. A request declaring `hubspot:deal-`

`fetch` (or `-update`) must be evaluated against exactly `hubspot-deal-update@1.0.0`, or it is rejected before any policy file is even loaded.

HubSpot (`packages/connector-hubspot/src/`): two capabilities, fetch and update a deal. `HubSpotSignalStateVerifier` (a separate file in the same package) independently confirms a caller-declared signal like "this deal's stage really is X" against HubSpot's own API rather than trusting the caller's assertion, the concrete example Chapter 5 (signal-state verification) points to.

GitHub (`packages/connector-github/src/`): fetch a PR's state, merge a PR. Uses a GitHub App JWT (`GitHubAppJwt.ts`) and an installation-token credential provider (`GitHubAppCredentialProvider.ts`, in `execution-gateway`) rather than a static personal access token.

Paytm (`packages/connector-paytm/src/`): exactly one capability, `paytm:refund`, by design. Its own doc comment is explicit that this is deliberate: "there is no `paytm:*` wildcard and no additional Paytm capability... declared here; adding one is a deliberate, separate milestone" (`PaytmCapabilities.ts:6-11`). Structurally different from the other three connectors in one respect: it never talks to Paytm's own API directly. `baseUrl` is required, not optional, because every request goes to a separate, trusted, out-of-process service (`parmana-paytm-agent`, a different repository, the "Pfinite" integration referenced elsewhere in this codebase's operational history), which is the only thing that ever holds a real Paytm merchant key (`PaytmCapabilities.ts:60-66`). The wire-level action string this remote service expects, `"paytm-refund"` (`PAYTM_AGENT_WIRE_ACTION`), is a pure transport-boundary translation; Parmana's own internal capability identity stays `"paytm:refund"` everywhere else and is never renamed to satisfy the remote service's naming convention (`PaytmCapabilities.ts:30-40`).

Slack (`packages/connector-slack/src/`): exactly one capability, `slack:post-message`, same "no wildcard, deliberate scope" discipline as Paytm (`SlackCapabilities.ts:6-11`).

Razorpay: a historical case study

A Razorpay connector (`RazorpayConnector`, `RazorpayRefundService`, `RazorpaySignalStateVerifier`, `RazorpayDailyRefundLedger`, `RazorpayCumulativeRefundLedger`, `RazorpaySettlementProcessor`, and supporting files under `packages/connector-sdk/src/connectors/razorpay/`) was built, ran in production, and was **deliberately removed from this codebase in its entirety on 2026-08-12**, commit `d8a6ded` ("Add HubSpot integration evidence and update TRL assessment"). `git log --all` for any of those paths shows no file at current `HEAD`, confirmed directly, not inferred.

Being honest about what the record does and doesn't say: `docs/VERIFICATION-GAPS.md`'s own "stale-narrative notice" (added 2026-08-24, search for "blocks-pilot") documents the removal happened and exactly when, and confirms `HubSpotSignalStateVerifier` and the HubSpot/GitHub capability pairs are what's actually reachable in production today in its place. It does not state an explicit business reason for the removal beyond the coincidence with the HubSpot integration landing in the same commit. Three residual database tables (`razorpay_webhook_events`, `razorpay_webhook_audit_events`, `razorpay_daily_refund_reservations`) survived the code removal until this session, 2026-09-16, when they were finally dropped (`supabase/migrations/20260916120000_drop_razorpay_tables.sql`) as pure schema cleanup, over a month after the connector code itself was gone.

The practical lesson this case study leaves: `docs/VERIFICATION-GAPS.md` entries that predate 2026-08-12 and describe Razorpay-specific mechanisms as "wired into production" are accurate history, not current fact, and the document itself now flags this explicitly rather than silently going stale.

How to validate this yourself

- `packages/connector-sdk/src/ConnectorTypes.ts`, the `Connector` contract itself.
- `packages/connector-hubspot/src/`, `connector-github/src/`, `connector-paytm/src/`, `connector-slack/src/`, each connector's capability metadata and DTOs.
- `packages/execution-gateway/src/connector-execution/Gateway{HubSpot,GitHub,Paytm,Slack}Adapter.ts`, the actual executable adapters.
- `packages/capability-registry/src/CapabilityPolicyBinding.ts`, the canonical capability-to-policy table.
- `packages/api/src/bootstrap/create{HubSpot,GitHub,Paytm,Slack}Connector.ts` and `assertConnectorCapabilitiesBound.ts`, how connectors get registered at startup, and the startup assertion that every declared capability has a canonical policy binding.

- Integration tests: `packages/api/tests/integration/hubspot-deal-update.integration.test.ts`, `github-pr-merge.integration.test.ts`, `paytm-refund.integration.test.ts`, each connector's real, HTTP-boundary proof, including its mock server (`MockHubSpotServer.ts`, `MockGitHubServer.ts`, `MockPaytmConnectorServer.ts`, `MockSlackServer.ts`) for hermetic testing without real vendor credentials.

Integration requirements

Each connector needs its own credentials configured, only when actually used:

- HubSpot: `HUBSPOT_PRIVATE_APP_TOKEN` (defaults to HubSpot's real API; tests point `baseUrl` at a local mock instead).
- GitHub: `GITHUB_APP_ID`, `GITHUB_INSTALLATION_ID`, `GITHUB_APP_PRIVATE_KEY`.
- Paytm: a `baseUrl` pointing at a deployed `parmana-paytm-agent` instance, required, with no default, since there is no direct-to-Paytm path at all.
- Slack: (see `createSlackConnector.ts` for the exact env var; defaults to Slack's real Web API base URL otherwise).

A connector with no credentials configured is simply not registered at startup rather than registered in a broken state, each `create*Connector.ts` bootstrap file logs an `*_connector_unavailable` event naming the missing configuration, visible in this codebase's own startup logs.

Chapter 13: The Storage Layer

What It Is

The storage layer is the set of repositories that give every durable Parmana concept (a business transaction, an execution trust record, a refusal record, a pending policy change, a policy approval record, and, as of 2026-09-16, live policy content itself) a place to live beyond a single process's memory. It is accessed through one interface, `StorageProvider` (`packages/storage/src/StorageProvider.ts`), with two real implementations selected at startup: an in-memory one for tests and local development, and a Postgres-backed one for anything real.

Why It Was Built

Parmana's whole value proposition depends on durable, independently verifiable evidence. A signed execution trust record that only exists in process memory and disappears on restart proves nothing to anyone after the fact. The storage layer exists to make every artifact this system produces outlive the process that produced it, without forcing every package that needs to persist something to independently decide how.

How It Works

The interface

`StorageProvider` exposes five repositories:

```
export interface StorageProvider {
  readonly businessTransactions: BusinessTransactionRepository;
  readonly trustRecords: ExecutionTrustRecordRepository;
  readonly refusalRecords: RefusalRecordRepository;
  readonly pendingPolicyChanges: PendingPolicyChangeRepository;
  readonly policyChangeApprovalRecords: PolicyChangeApprovalRecordRepository;
}
```

(`packages/storage/src/StorageProvider.ts:15-42`)

Two implementations, chosen by `StorageFactory`

`StorageFactory.create({ provider })` (`packages/storage/src/StorageFactory.ts`) switches on the configured provider:

Provider value	Class returned	Notes
`"memory"`	`MemoryStorageProvider`	Five in-memory repository classes, nothing external touched.
`"supabase"`	`SupabaseStorageProvider`	Requires `DATABASE_URL`; throws immediately if missing, rather than constructing a pool that would only fail on first use.
`"postgres"`	throws `"Postgres storage provider not implemented."`	Reserved name, not built.
`"sqlite"`	throws `"SQLite storage provider not implemented."`	Reserved name, not built.

`StorageFactory.createFromEnvironment()` is the real entry point every caller actually uses. It has one hardcoded override: under `NODE_ENV=test`, it always returns `MemoryStorageProvider`, regardless of what `PARMANA_STORAGE` is set to. This exists because of a real incident (G-15, `docs/VERIFICATION-GAPS.md`): this method used to construct whatever `PARMANA_STORAGE` named as a pure import-time side effect, which crashed test collection with supabase-js's generic `"supabaseUrl is required."` the moment `SUPABASE_*` was unset, regardless of what any individual test actually needed.

Lazy construction, everywhere that matters

`packages/api/src/repositories.ts` wraps every repository export in a `Proxy` via a local `lazyRepository()` helper, so importing `repositories.ts` (which every route file transitively does) never itself constructs a live Supabase client. Only the first actual method call on a repository triggers `StorageFactory.createFromEnvironment()`.

This exact discipline was violated, then fixed, in this codebase on 2026-09-16. The first version of `packages/api/src/application.ts`'s new `policyRepository` export constructed `SupabasePolicyRepository(PostgresPoolFactory.create())` directly at module scope:

```
export const policyRepository: PolicyRepository =
  process.env.NODE_ENV !== "test" && config.storage.provider !== "memory"
    ? new SupabasePolicyRepository(PostgresPoolFactory.create())
    : new FilePolicyRepository(config.policy.directory);
```

Because `PostgresPoolFactory.create()` is a process-wide singleton (see below), this opened a real connection against whatever `DATABASE_URL` happened to be set at the moment `application.ts` was first imported, before any caller had a chance to configure it differently. This broke `examples/tutorials/89-readiness-probe/run.ts`'s own "genuinely unreachable database" scenario: the tutorial tries to swap in a bad `DATABASE_URL` for one specific test case, but the pool had already been created against the real, working one by the time that scenario ran, so the readiness check kept reporting `READY` when it should have reported `NOT_READY`. The fix (`packages/api/src/application.ts`, current version) wraps `policyRepository` in the identical `Proxy`-based lazy pattern `repositories.ts` already uses, deferring construction to the first real `load()` / `save()` / `listAll()` call.

Direct Postgres, not PostgREST

`PostgresPoolFactory` (`packages/storage/src/postgres/PostgresPoolFactory.ts`) is a lazy, process-wide singleton `pg.Pool`. Every Supabase-backed repository in this codebase connects through it directly, not through `supabase-js`'s REST-based client (PostgREST). The class that used to provide that client, `SupabaseClientFactory`, was deleted 2026-09-09 (`docs/VERIFICATION-GAPS.md` G-34) after `PostgresPoolFactory`'s own pattern had already replaced it everywhere it was still used. This removes an entire layer (PostgREST) from the failure modes of every table these repositories touch, an incident class this codebase's own history names explicitly (see `SupabaseCallerAuditSink`'s own comments for the originating case).

As of 2026-09-16, `PostgresPoolFactory.create()`'s `Pool` is constructed with `connectionTimeoutMillis: 5000`. Before that, an unreachable or misconfigured `DATABASE_URL` hung a connecting query indefinitely rather than failing fast, found via the same readiness-probe tutorial above, whose "genuinely unreachable database" scenario never returned before this was added.

The rate-limit store, a Postgres-backed class with no repository interface

`PostgresRateLimitStore` (`packages/storage/src/postgres/PostgresRateLimitStore.ts`) is not part of `StorageProvider`, it implements `express-rate-limit`'s own `Store` contract directly (structurally, via a locally-defined `RateLimitStoreLike` interface, so `@parmana/storage` never depends on the `express-rate-limit` package itself). It exists to close a fleet-wide gap: `express-rate-limit`'s default `MemoryStore` counts per-process, so on a horizontally-scaled deployment the effective ceiling for a caller becomes `limitPerMinute * machineCount` rather than the configured limit. See Chapter 16 for the full rate-limiting story, including a real `ERR_ERL_STORE_REUSE` bug this class's `prefix` field exists to resolve.

An orphaned repository, honestly

`packages/storage/src/memory/MemoryPolicyRepository.ts` exists, is exported from `@parmana/storage`'s public index, and implements `PolicyRepository`. As of this writing, it is not referenced anywhere in `MemoryStorageProvider`, `StorageProvider`, or `application.ts`, the actual in-memory policy path uses `FilePolicyRepository` against a scratch or configured directory instead. This is the same category of finding the existing `docs/book/` chapter 16 already documents for `@parmana/receipt` / `@parmana/replay`: real, presumably tested code that is not part of the live wiring. Confirm this yourself before relying on it (see "How to Validate" below), it may simply be unused, or it may be a recent addition not yet wired in.

The schema itself

`supabase/migrations/*.sql` is the real source of truth for every table these repositories read and write. As one directly-verified example, the `policies` table added 2026-09-16 (`supabase/migrations/20260916060000_add_policies_table.sql`) is

a two-column-key table:

```
CREATE TABLE IF NOT EXISTS policies (
    policy_name TEXT NOT NULL,
    policy_version TEXT NOT NULL,
    content_json JSONB NOT NULL,
    updated_at TIMESTAMPTZ NOT NULL DEFAULT now(),
    PRIMARY KEY (policy_name, policy_version)
);
```

Note this table is read and written by `SupabasePolicyRepository` (`packages/policy/src/SupabasePolicyRepository.ts`), which lives in the `@parmana/policy` package, not `@parmana/storage`, it is constructed directly in `application.ts`, not exposed through `StorageProvider`. This is a real architectural asymmetry: every other Postgres-backed repository in this codebase goes through the `StorageProvider` facade; policy content does not. Neither the code nor any comment currently explains why this one repository sits outside the facade rather than being added to it as a sixth field, treat this as an open question rather than an assumed design decision.

`docs/architecture/DATABASE_SCHEMA_REFERENCE.md` is a detailed, table-by-table reference covering every table in this schema; this chapter verified one table directly against the migration SQL rather than only citing that document, and recommends the same discipline for any other table you need to rely on.

How It Enables Things, With a Concrete Example

`examples/tutorials/116-supabase-policy-repository/run.ts` exercises `SupabasePolicyRepository` directly against a minimal fake `pg.Pool` (no real network), proving `save()` then `load()` round-trips exactly, `load()` on a missing entry throws `PolicyNotFoundError`, and `listAll()` reports what was saved, the same contract `FilePolicyRepository` upholds, just backed by a table instead of the filesystem.

`examples/tutorials/115-per-limiter-rate-limit-stores/run.ts` exercises `PostgresRateLimitStore` against a similar fake `pg.Pool`, proving the real `ERR_ERL_STORE_REUSE` failure mode and its fix (Chapter 16 covers this fully).

How to Validate This Yourself

- `packages/storage/src/StorageProvider.ts`, `StorageFactory.ts`, the interface and the selection logic.
- `packages/storage/src/memory/MemoryStorageProvider.ts`, `packages/storage/src/supabase/SupabaseStorageProvider.ts`, the two real implementations.
- `packages/storage/src/postgres/PostgresPoolFactory.ts`, the shared connection pool, its own comments name the real incidents that shaped it.
- `packages/api/src/repositories.ts` and `packages/api/src/application.ts`, the lazy construction pattern, and the place it was once violated.
- `supabase/migrations/*.sql`, the real, current schema. Read the actual files; a reference document can drift, the migration files cannot (they are the thing that ran).
- `packages/storage/tests/unit/postgres-pool-factory.test.ts`, confirms the exact `Pool` constructor arguments, including `connectionTimeoutMillis`.
- Search the codebase yourself for `MemoryPolicyRepository` usage before trusting this chapter's claim that it is orphaned; that kind of claim is exactly the kind that goes stale fastest.

Integration Requirements

Variable	Required when	Effect
`PARMANA_STORAGE`	Always (defaults to `memory`)	`memory` or `supabase`; selects the `StorageProvider`.
`DATABASE_URL`	`PARMANA_STORAGE=supabase`	Direct Postgres connection string. `StorageFactory.create()` throws immediately if this is missing under `supabase`, naming both knobs, rather than deferring to a confusing `PostgresPoolFactory` error later.
`NODE_ENV=test`	Test runs	Forces `MemoryStorageProvider` regardless of `PARMANA_STORAGE`.

Chapter 14: The API and HTTP Boundary

What It Is

`packages/api/src/app.ts`'s `createApp()` is the single function that assembles every HTTP route this system exposes into one Express application, in a specific, deliberate mounting order. Everything an external caller can reach, from an unauthenticated health check to a signed execution request, passes through this one file's wiring.

Why It Was Built

An HTTP API needs a consistent, auditable answer to two questions for every route: does this route require a caller identity, and what happens when something goes wrong. `app.ts` answers both in one place rather than leaving each route handler to decide independently, which is what makes it possible to state precisely (not just claim) which routes are public and why.

How It Works

Mounting order, and why it matters

Routes are mounted in this order, and the order is load-bearing: caller-auth middleware (`createCallerAuthMiddleware`) is only added to the pipeline partway through, so everything mounted before it is unauthenticated by construction, not by an exception carved out of a uniform rule.

Path	Auth required	Why
`GET /health`	No	Liveness probe; a PaaS orchestrator has no API key to present.
`GET /ready`	No	Readiness probe (see Chapter 19); same reasoning as `/health`.
`GET /openapi.yaml`, `/openapi.json`	No	A caller cannot discover how to get a key from a spec it is not allowed to read.
`GET /api-manifest.json`	No	Self-description of SDK versions, same public-discovery reasoning.
`/documentation`, `/reference`	No	API documentation consumers.
`POST /refusal/verify`	No	RFC-0021: unauthenticated, third-party signature verification, not a data-access route.
`POST /audit/verify`	No	Same category as above, over `caller_audit_events` instead of Refusal Records, pure signature-over-bytes, no database lookup.
`/keys/:keyId`, `/.well-known/jwks.json`	No	Public-key discovery (PQC audit RED-2); a third party verifying a signature cannot be required to already hold a credential.
, caller-auth middleware mounted here if `callerAuth !== "disabled"` ,		
`GET /`	No (mounted before, but trivial)	Returns `{ name: "Parmana", status: "UP" }`.
`/version`	Yes (if auth enabled)	
`/callers/me`	Yes	Caller self-lookup: identity and exactly what it's authorized to do.
`/execute`	Yes	Rate-limited by `callerId` (Chapter 16); the limiter itself is only mounted when caller-auth is enabled, since there is no caller identity to key off otherwise.
`/verify`, `/verification`	Yes	

Path	Auth required	Why
`/refusal`	Yes	Ownership-scoped lookup by ID (the unauthenticated verify route above is separate).
`/receipt` , `/receipt/latest`	Yes	
`/transactions`	Yes	
`/policies`	Yes	Mounted twice: once for the policy-content routes (`policies.ts`), once for the Policy Governance pending-changes routes (`pending-policy-changes.ts`), their path shapes don't collide.
`/trust-records`	Yes	
`/replay`	Yes	
, error handler mounted last ,		

(`packages/api/src/app.ts:154-347`)

Error mapping

`createErrorHandler()` (`packages/api/src/middleware/error-handler.ts`) is the single place a thrown domain error becomes an HTTP response:

Thrown	Status	Notes
Malformed/oversized JSON body (`express.json()` , before any route handler)	400 / 413	Best-effort audited even though no caller identity exists yet (G-29), deliberately fail-**open** here, unlike every other caller-audit write, since the correct rejection has already happened by the time this fires.
`BusinessTransactionValidationError` , `PolicyValidationError` , `SignalValidationError`	400	
`PolicyNotFoundError`	404	
`DuplicateBusinessTransactionError`	409	
`NonceAlreadyConsumedError`	`error.status` (its own code)	Distinguished from a generic `RuntimeError` so a caller can tell "this already ran" apart from "something broke" without string-matching a 500 body.
Any `RuntimeError`	`error.status`	
Anything else	500	Logged via `console.error` , response body is the generic `"Internal Server Error"` , no internal detail leaked.

(`packages/api/src/middleware/error-handler.ts:95-201`)

The full route inventory

Every route module lives in `packages/api/src/routes/` :

File	Mounted at	Purpose
`health.ts`	`/health`	Liveness.
`ready.ts`	`/ready`	Readiness (Chapter 19).
`openapi.ts` / `openapi-json.ts`	`/openapi.yaml` / `/openapi.json`	Machine-readable API spec.
`api-manifest.ts`	`/api-manifest.json`	SDK version self-description.
`documentation.ts` / `reference.ts`	`/documentation` / `/reference`	Human-readable docs.
`refusal-verify.ts`	`/refusal/verify`	Public signature verification of a Refusal Record.

File	Mounted at	Purpose
`audit-verify.ts`	`/audit/verify`	Public signature verification of an audit event.
`keys.ts`	`/keys/:keyId`, `/.well-known/jwks.json`	Public key discovery.
`callers-me.ts`	`/callers/me`	Caller self-lookup.
`execute.ts`	`/execute`	Submit and execute a Business Transaction.
`verify.ts` / `verify-get.ts`	`/verify` / `/verification`	Verification of an executed transaction.
`refusal-get.ts`	`/refusal`	Ownership-scoped Refusal Record lookup.
`receipt.ts` / `receipt-get.ts`	`/receipt` / `/receipt/latest`	Receipt issuance and lookup.
`transactions.ts`	`/transactions`	Business Transaction create-and-execute (parity with `/execute`).
`policies.ts`	`/policies`	Policy content read/validate routes.
`pending-policy-changes.ts`	`/policies`	Maker-checker propose/list/approve/reject (Chapter 7).
`trust-records.ts`	`/trust-records`	Bulk export.
`replay.ts`	`/replay`	Replay a prior execution.
`version.ts`	`/version`	
`findOpenApiSpecFile.ts`	(helper, not a route)	Locates the built OpenAPI spec file on disk.

`trust proxy`

`app.set("trust proxy", 1)` (`packages/api/src/app.ts:147`) trusts exactly one proxy hop , this codebase's actual deployment fronting (Fly.io's edge, per `fly.toml`'s `force_https`). Without this, `req.ip` and `req.protocol` / `req.secure` reflect the proxy's own connection to the process, not the original client's.

How It Enables Things, With a Concrete Example

`examples/tutorials/102-distinguishable-http-status` demonstrates the `NonceAlreadyConsumedError` vs. generic-`RuntimeError` distinction in the error handler directly, a policy denial and a replay attempt against the same endpoint return genuinely different, distinguishable status codes rather than both collapsing into an opaque failure.

How to Validate This Yourself

- `packages/api/src/app.ts`, the actual mounting order; read it top to bottom rather than trusting the table above once this file changes.
- `packages/api/src/middleware/error-handler.ts`, the full error-to-status mapping.
- `packages/api/src/routes/*.ts`, one file per route group.
- `packages/api/tests/integration/*.integration.test.ts`, most route groups have a corresponding integration test that exercises the real HTTP path, not just the handler function in isolation.

Integration Requirements

`createApp()` requires a `CallerAuthOption`, either a real `{ authenticator, auditSink }` pair or the literal string `"disabled"` (used only for local development and the tutorial suite; there is deliberately no default, so omitting this choice is not possible). Optional: `RateLimitOption` (Chapter 16), `stepUpVerifier` and `policyChangeApprovalService` (both required in effect, though optional at the type level, once caller-auth is enabled and the Policy Governance approve/reject endpoints are actually reachable, see Chapter 7).

Chapter 15: Caller Authentication and Scoping

What It Is

The layer that decides whether an HTTP request is entertained at all, who it came from, and what that identity is actually allowed to do (which principal it may assert, which capabilities it may invoke). It runs before a Business Transaction is even constructed and is entirely independent of policy evaluation and gateway attestation, which run later and answer different questions.

Why It Was Built

Without an identity layer, authority/authorization fields on a submitted transaction are purely caller-declared and never cross-checked against who is actually calling, any caller holding any valid API key could claim to be any human or role in the resulting signed trust record. Caller authentication and scoping close that gap at the door, before anything downstream has a chance to trust a claim it shouldn't.

How It Works

Authentication: `StaticKeyAuthenticator`

`packages/api/src/auth/StaticKeyAuthenticator.ts` authenticates against a static, pre-hashed set of API keys (`ApiKeyEntry[]`, from `PARMANA_API_KEYS`). Keys are never held in plaintext, only a SHA-256 hash of each is compared, via `timingSafeEqual` against the hash bytes (never the raw key), so a leaked config never yields a usable secret and comparison timing never leaks information about a partial match.

Multiple entries may share the same `callerId`, this is how key rotation works: add the new key's hash, keep the old one active during migration, then remove the old entry to revoke it, with no downtime and no code change.

`ApiKeyEntry` (`packages/shared/src/config/ApiKeyEntry.ts`):

```
export interface ApiKeyEntry {
  readonly callerId: string;
  readonly keyHash: string;
  readonly credentialHolderType?: AuthorityType;
  readonly allowedPrincipalIds?: readonly string[];
  readonly allowedCapabilities?: readonly string[];
  readonly stepUpPublicKey?: string;
}
```

`credentialHolderType` is operator-declared metadata set only at issuance time, distinct from `Authority.authorityType`, which is caller-declared and lives on the Business Transaction itself describing the business action's asserted authority. The two must not be conflated: one describes who was handed the credential, the other describes what a specific request claims. `isHumanCaller.ts` is the only consumer of `credentialHolderType`, and treats every value other than exactly `AuthorityType.USER`, including `undefined`, as non-human, fail-closed by default.

The middleware itself

`createCallerAuthMiddleware()` (`packages/api/src/middleware/caller-auth.ts`) extracts a bearer token, authenticates it, and on success attaches `callerId`, `callerAllowedPrincipalIds`, `callerAllowedCapabilities`, `callerCredentialHolderType`, and `callerStepUpPublicKey` onto the Express `Request` object (a local type augmentation, same pattern `@parmana/envelope-verifier`'s own Express typing uses). On failure it returns 401 with a `WWW-Authenticate: Bearer` header, and either way it records a `caller.rejected` or `caller.authenticated` audit event through `recordCallerAuditEvent` before proceeding, **fail-closed**: if the audit write itself fails, the request is rejected with `AuditUnavailableError` (503) rather than proceeding unaudited. This applies to both outcomes, not just denials, a perfectly valid credential whose `caller.authenticated` write fails also gets 503, not the 200 it would otherwise get.

Scoping: principal and capability

Two independent checks, deliberately opposite in their fail-closed default:

`isPrincipalAllowed()` (`packages/api/src/auth/isPrincipalAllowed.ts`) decides whether a caller may assert a given `authority.principalId`. Default (no `allowedPrincipalIds` configured): a key may only assert **itself**, `principalId` must equal `callerId` exactly. An unconfigured key proves its own identity and nothing more.

`isCapabilityAllowed()` (`packages/api/src/auth/isCapabilityAllowed.ts`) decides whether a caller may invoke a given capability (the transaction's `intent.action`) at all. Default (no `allowedCapabilities`, or an empty list): **every** capability is denied. There is no meaningful "may invoke its own capability" fallback the way "may only assert itself" is for principals, an unconfigured key is authorized to invoke nothing until explicitly granted. The literal string `"*"` in `allowedCapabilities` is an explicit, auditable wildcard, never an implicit default.

`PARMANA_AUTH_DISABLED`

A deployment can set `PARMANA_AUTH_DISABLED=true` to accept every request with no caller authentication at all. This is surfaced not just as a startup log line but as a field on `GET /ready`'s own response body (`packages/api/src/routes/ready.ts:44-52`):

```
{
  "authDisabled": true,
  "warning": "PARMANA_AUTH_DISABLED=true -- this deployment is accepting requests with no caller authentication.
}
```

This exists because a log line is easy to miss after the fact in a log-aggregation tool; a field on the readiness probe every PaaS orchestrator already polls every 30 seconds is something an operator's own monitoring can assert and alert on directly.

How It Enables Things, With a Concrete Example

`examples/tutorials/101-fail-closed-caller-audit-writes/run.ts` proves the audit-write fail-closed guarantee directly: it simulates the storage outage `SupabaseCallerAuditSink` would surface, and shows both `caller.rejected` and `caller.authenticated` outcomes react identically (503, not their otherwise-normal status) when the audit write itself fails. There is no retry, buffering, or queueing, a failure fails closed immediately, once, per request.

`packages/api/tests/integration/caller-scoping.integration.test.ts`, `caller-principal-scoping.integration.test.ts`, and `caller-capability-scoping.integration.test.ts` exercise the two scoping checks at the real HTTP boundary, including a documented IDOR-regression suite (blocking one caller from reading another's transactions, receipts, trust records, and refusal records by ID).

How to Validate This Yourself

- `packages/api/src/middleware/caller-auth.ts`, the middleware itself.
- `packages/api/src/auth/StaticKeyAuthenticator.ts`, `hashApiKey.ts`, authentication.
- `packages/api/src/auth/isPrincipalAllowed.ts`, `isCapabilityAllowed.ts`, `isHumanCaller.ts`, `isOwnedByCaller.ts`, the four scoping/identity predicates.
- `packages/shared/src/config/ApiKeyEntry.ts`, the credential shape.
- `packages/api/src/routes/ready.ts`, the `authDisabled` warning field.
- `packages/api/tests/integration/caller-*.integration.test.ts`, the real HTTP-level proof for every claim above.

Integration Requirements

Variable	Effect
`PARMANA_API_KEYS`	JSON array of `ApiKeyEntry` objects, the entire credential set for a deployment.

Variable	Effect
`PARMANA_AUTH_DISABLED`	`true` disables caller-auth entirely. Never set this in a real deployment; the `/ready` endpoint will say so loudly if you do.

Provisioning a new credential: `scripts/generate-api-key.ts` (see Chapter 7 for the step-up-keypair variant used by Policy Governance checkers).

Chapter 16: Rate Limiting

What It Is

Two independent `express-rate-limit` limiters, one for `POST /execute` (keyed by authenticated caller identity) and one for `GET /health` / `GET /ready` (keyed by IP, far more permissive), each backed by either an in-process store or a durable, fleet-wide Postgres store depending on deployment configuration.

Why It Was Built

Two separate problems. First, ordinary abuse/overload protection on the one endpoint that actually does real work (signing, storage writes). Second, a real production-readiness gap found 2026-09-10: `express-rate-limit`'s default `MemoryStore` counts per-process, so on a horizontally-scaled deployment the effective ceiling for a caller becomes `limitPerMinute * machineCount` rather than the configured, intended limit. `PostgresRateLimitStore` exists to close that gap for any deployment where it matters.

How It Works

Two limiters, deliberately different keying

`createExecuteRateLimiter()` (`packages/api/src/middleware/rate-limit.ts:71-90`) keys by `req.callerId`, not IP, a design-partner integration commonly calls from a shared backend IP, where an IP-keyed limit would either starve every caller behind it or be too loose to mean anything. It is only ever mounted when caller authentication is enabled (`app.ts`), since `req.callerId` does not exist otherwise, and it runs ahead of the route handler, so a rejected request never reaches policy evaluation, signing, or storage, nothing is consumed for a request this middleware rejects.

`createHealthReadyRateLimiter()` (`:103-115`) keys by IP (the package default, IPv6-safe in `express-rate-limit` v8), and is deliberately far more permissive, these routes are cheap and legitimately polled on a fixed interval by PaaS health-check infrastructure (`fly.toml`'s own check runs every 30 seconds per machine); this limiter must never be tight enough to throttle that.

Both share one 429 response shape via `rateLimitHandler()`, which sets `Retry-After` explicitly from the store's own `resetTime` rather than trusting a package default header.

The real bug: one shared Store, two limiters

Found 2026-09-15, during the same session as the AWS KMS migration: `createApp()`'s first version constructed a single `PostgresRateLimitStore` and passed it to both limiters. `express-rate-limit` v8's own documented contract disallows this, a `Store` instance must not back more than one limiter, and the library throws `ERR_ERL_STORE_REUSE` the moment a second limiter's `init()` runs against an already-initialized store. Passing one shared instance to both `createExecuteRateLimiter` and `createHealthReadyRateLimiter` crashed every request on a fresh cold start.

A documented correction, not silently fixed: it was first assumed this crashed the whole request (`docs/VERIFICATION-GAPS.md`, `docs/operations/2026-09-15-kms-migration-troubleshooting-guide.md` item 5), inferred from seeing the `ERR_ERL_STORE_REUSE` log line next to a real HTTP 500. Reading `express-rate-limit`'s actual installed source shows every validation is wrapped in a try/catch that only logs the violation and never re-throws, the reuse warning is real, but harmless on its own. The 500 that request actually returned was caused by a completely different, unrelated bug (see `examples/tutorials/114-signing-verification-key-agreement`). The fix (two separate `Store` instances, each with a distinct `prefix`) is still correct and worth having, independent of the corrected causal story, it is the library's own documented usage contract, and a stricter `validate` config in a future version could make this fatal for real.

The fix itself: `RateLimitOption` (`packages/api/src/app.ts:71-91`) carries two distinct fields, `executeStore` and `healthStore`, never one shared `store`. `createRateLimitStore()` (`packages/api/src/bootstrap/createRateLimitStore.ts`) is called once per limiter, each with its own `prefix` (e.g. `"execute:"` and `"health:"`), so their counters can never collide in the shared `rate_limit_counters` table even though both instances share the same underlying `Pool` (`PostgresPoolFactory.create()` is itself a singleton, so calling this twice does not open a second connection pool).

`PostgresRateLimitStore.prefix` (`packages/storage/src/postgres/PostgresRateLimitStore.ts:80`) is deliberately **public**, not a private implementation detail, this is `express-rate-limit`'s own documented `Store.prefix` contract, and naming the field exactly `prefix` is what lets the library's own reuse/double-count validation recognize two instances with different prefixes as legitimately distinct.

Three-way store selection

`createRateLimitStore(prefix)` returns:

Condition	Returns
`NODE_ENV=test`	`undefined`, falls back to `express-rate-limit`'s own in-process `MemoryStore`.
No `DATABASE_URL`	`undefined`, with a loud `console.warn` naming the exact tradeoff. Deliberately **not** fail-closed.
`DATABASE_URL` configured	`new PostgresRateLimitStore(PostgresPoolFactory.create(), prefix)`, durable, fleet-wide.

The "no `DATABASE_URL`" case is deliberately not fail-closed the way this codebase's nonce store (`createNonceStore.ts`) is: an in-process rate limiter is a real, working control on a single instance (this deployment's actual current shape, `fly.toml` pins `min_machines_running = 1`), just not fleet-wide accurate the moment a second machine joins. A missing nonce store means replay protection silently vanishes, a security bypass; a missing shared rate-limit store means the ceiling is merely looser than configured, not absent. Refusing to start over that would break every single-instance and local deployment that works correctly today.

How It Enables Things, With a Concrete Example

`examples/tutorials/115-per-limiter-rate-limit-stores/run.ts` reproduces the real `ERR_ERL_STORE_REUSE` warning directly (Scenario 1: one shared store, logged but not thrown) and confirms the fix (Scenario 2: two prefixed stores, no warning at all), hermetically, against a minimal fake `pg.Pool`, no real database needed. Its own README documents the corrected diagnosis explicitly, kept visible rather than quietly rewritten.

`packages/api/tests/integration/rate-limit.integration.test.ts` exercises both limiters at the real HTTP boundary: normal traffic passing through, a rate-limited caller not blocking a different caller, a 429 with a correct `Retry-After` header and no execution side effects, and confirms the limiter is entirely skipped when caller-auth is disabled.

How to Validate This Yourself

- `packages/api/src/middleware/rate-limit.ts`, both limiter factories.
- `packages/api/src/bootstrap/createRateLimitStore.ts`, the three-way selection logic, with its own comment stating the exact tradeoff for each branch.
- `packages/storage/src/postgres/PostgresRateLimitStore.ts`, the durable store, including the `prefix` field's own doc comment explaining exactly why it must be named that and be public.
- `packages/api/src/app.ts:71-91, 149-171, 248-264`, where both limiters are actually constructed and mounted.
- `examples/tutorials/115-per-limiter-rate-limit-stores/README.md`, the full incident account, including the corrected diagnosis.

Integration Requirements

Variable	Effect
`RATE_LIMIT_EXECUTE_PER_MINUTE`	Overrides the default execute-limiter ceiling (`DEFAULT_EXECUTE_PER_MINUTE = 30`).
`RATE_LIMIT_HEALTH_PER_MINUTE`	Overrides the default health/ready-limiter ceiling (`DEFAULT_HEALTH_PER_MINUTE = 300`).
`DATABASE_URL`	If set, both limiters use the durable, fleet-wide `PostgresRateLimitStore` instead of per-process memory.

Chapter 17: Audit and Evidence Trails

What it is

Parmana keeps several distinct, signed, durable trails of what happened: refusal records for rejected policy decisions, caller audit events for every authentication and authorization decision, execution audit events for the execution control lifecycle, and an explicit evidence anchor that ties a trust record's policy content, governance status, and connector evidence together into one checkable pointer. These are separate mechanisms with separate scopes, not one generic "audit log," and each exists to answer a specific question an auditor might ask.

Why it was built

An approval trail is only half the story. A system that only records what it approved cannot prove it correctly rejected everything it should have rejected, cannot prove a caller's authentication decisions were consistent over time, and cannot prove that a trust record's policy content, its governance status, and the connector evidence backing it are genuinely linked rather than merely sitting next to each other in the same object. Each trail here closes one of those specific gaps, added incrementally as each gap was found.

How it works

Refusal Records (RFC-0021)

`RefusalRecord` (`packages/shared/src/domain/refusal-record.ts`) is the durable, signed, independently verifiable REJECT-path counterpart to `ExecutionTrustRecord`. Its own doc comment is explicit about scope: it covers `PolicyEngine.evaluate` REJECTs and `SignalIntentBinder` binding-violation REJECTs only, not every kind of rejection this system can produce. Caller-auth failures and webhook signature failures are a separate, unsigned audit-sink milestone (see "Caller Audit Events" below), RFC-0021's own Non-Goals section draws this line. Unlike `ExecutionTrustRecord`, a `RefusalRecord` is not an append-only aggregate with sub-histories: a refusal is a single terminal event, so there is at most one per `businessTransactionId`.

The record carries the exact rejected `Decision` (not summarized or reconstructed, the same `Decision` object `RuntimeEngine` already built), the `evaluatedIntent` snapshot, and, only when the rejection came from `SignalIntentBinder`, `bindingViolations`. `RefusalRecordBuilder` (`packages/runtime/src/RefusalRecordBuilder.ts`) constructs and hashes the draft, then `RefusalCrypto` (`packages/crypto/src/RefusalCrypto.ts`) signs it, deliberately reusing the same signing stack and `DEFAULT_KEY_ID` as `VerificationCrypto`, so approvals and refusals share one root of trust (RFC-0021 §2), not two keys to manage. `RefusalCrypto.canonicalRecord()` excludes the signature itself from what gets hashed and signed, the same discipline every other signed artifact in this codebase follows.

Caller Audit Events

`SupabaseCallerAuditSink` (`packages/api/src/auth/SupabaseCallerAuditSink.ts`) durably records every caller-authentication decision: success, failure, capability checks, principal checks. Its own doc comment traces its own history precisely: it closes G-13 (a prior in-memory-only sink lost every event on restart), and four of its columns were each added in a separate, dated migration as a distinct milestone landed, `capability` (2026-08-12, capability-scoping), `principalId` (2026-08-16, principal-scoping), `severity` (2026-08-18, policy-governance), `businessTransactionId` (2026-08-24, G-29 structural-validation audit). Every event is signed at write time (`AuditEventCrypto`), before any storage-only field like an insert timestamp is added, a plain durable row could otherwise be altered by anyone with direct database access with no way to detect it.

One real, named operational detail worth knowing if you ever touch this class: it writes via a direct Postgres connection (`PostgresPoolFactory`), not `supabase-js`, because of a confirmed Supabase-side bug (ticket SU-437429) where PostgREST's schema cache refused to see the `signature_json` column. This is documented in the class's own comment as a temporary workaround at the PostgREST layer specifically, not a problem with the database or this codebase's own schema.

Failure semantics are explicit and deliberately unchanged from the sink this replaced: `record()` is an unguarded `await` in the caller-auth middleware with no try/catch around it. A write failure here rejects the promise exactly like any other failed

Supabase insert elsewhere in this codebase; it is the caller's existing (unmodified) behavior that decides what happens next, documented rather than silently changed.

Execution Audit Events

`SupabaseExecutionAuditSink` (`packages/storage/src/supabase/SupabaseExecutionAuditSink.ts`) covers the execution-control lifecycle separately from caller authentication. Per `docs/architecture/DATABASE_SCHEMA_REFERENCE.md`'s own note (verify this claim yourself if you touch this table, it is a real operational fact worth confirming against current deployment config, not just trusting a comment), the underlying `execution_audit_events` table is shared cross-repository with a separate `parmana-paytm-agent` repository, meaning this table's schema is a contract between two codebases, not something to change unilaterally from this one.

Evidence Anchor

`EvidenceAnchor` (`packages/shared/src/domain/evidence-anchor.ts`) is the newest of these mechanisms. Its own doc comment states plainly what it is and is not: not a new cryptographic guarantee (`policyContentHash`, the governance anchor, and connector evidence were already bound together implicitly, since all three already sit inside the same `ExecutionTrustRecord` that `trustRecordHash` / `signature` cover in full), what it adds is a single, explicitly-named, independently-computed pointer, `{policyContentHash, governanceAnchorStatus, connectorEvidenceHash, anchorHash}`, that an auditor can check without already knowing to reach into `transaction.policy` and `executions[].evidence.attributes.connector` separately and reconstruct the binding themselves. `anchorHash` is a `TrustRecordHasher` hash of the canonicalized triple, computed the same way every other artifact hash in this codebase is. `BusinessTrustRecordBuilder.buildEvidenceAnchor()` (`packages/runtime/src/`) builds it; every field is optional individually (absent when there was nothing to anchor), but `anchorHash` is always present.

Policy Governance Anchor Resolver

`PolicyGovernanceAnchorResolver` (`packages/api/src/governance/PolicyGovernanceAnchorResolver.ts`) performs three checks against `PolicyChangeApprovalRecordRepository`, in order: does an approval record exist for this `(policyName, policyVersion)` at all (`NO_APPROVAL_RECORD` if not); does its signature verify (`SIGNATURE_INVALID` if not); does its `contentHashAfter` match the policy content actually used for this decision (`CONTENT_MISMATCH` if not). A clean pass returns `VERIFIED`. Critically, this class is wired unconditionally into `RuntimeEngine`, with no feature flag, because it only ever records what it found, it never blocks execution. This is the deliberate opposite of `PolicyGovernanceExecutionVerifier` (the execution-time enforcement gate covered in Chapter 8/10's territory, not here), which has the same three checks but is gated behind `POLICY_EXECUTION_VERIFICATION_ENFORCED` precisely because a false positive there refuses a real execution. The class's own comment names this distinction explicitly and traces it to RFC-0022's precedent (`SignalStateVerifier` / `PolicyExecutionVerifier`): keep enforcement and evidentiary concerns in separate types even when their logic overlaps, rather than deduplicating into one shared helper with two different blast radii.

How it enables things, with a concrete example

`packages/api/tests/integration/refusal-record.integration.test.ts` and `packages/api/tests/integration/structural-validation-audit.integration.test.ts` exercise the refusal and caller-audit paths at the real HTTP boundary. No standalone tutorial in `examples/tutorials/` exercises `EvidenceAnchor` / `PolicyGovernanceAnchorResolver` directly as of this writing, their coverage lives in `packages/api/tests/unit/PolicyGovernanceAnchorResolver.test.ts` and two `packages/runtime/tests/e2e/runtime.e2e.test.ts` cases (result stamped correctly on a real execution; a resolver failure never blocks one). Said honestly: this is real, tested, production-wired code, just not yet demonstrated as a narrative tutorial the way policy governance itself is (Chapter 7).

How to validate this yourself

- `packages/shared/src/domain/refusal-record.ts`, `packages/runtime/src/RefusalRecordBuilder.ts`, `packages/crypto/src/RefusalCrypto.ts`
- `packages/api/src/auth/SupabaseCallerAuditSink.ts` and its migrations, named in its own comment, under `supabase/migrations/`
- `packages/storage/src/supabase/SupabaseExecutionAuditSink.ts`

- `packages/shared/src/domain/evidence-anchor.ts`, `packages/api/src/governance/PolicyGovernanceAnchorResolver.ts`
- `packages/api/tests/unit/PolicyGovernanceAnchorResolver.test.ts`, `packages/api/tests/integration/refusal-record.integration.test.ts`, `packages/api/tests/integration/structural-validation-audit.integration.test.ts`

Integration requirements

`DATABASE_URL` (direct Postgres connection) for `SupabaseCallerAuditSink` and `SupabaseExecutionAuditSink` to persist durably; without it, this codebase falls back to in-memory sinks that lose events on restart (correct for tests, not for production). No separate configuration is needed for `EvidenceAnchor` / `PolicyGovernanceAnchorResolver`, they are wired unconditionally whenever `RuntimeEngine` is constructed.

Chapter 18: Independent Verification

What it is

A third party with no access to a running Parmana instance at all, no API key, no network access to this deployment, should still be able to verify that a signed trust record, receipt, or refusal record is genuine and unmodified, given only the artifact itself and the relevant public key. This chapter covers the offline verification path, how a public key is discovered in the first place, and an honest assessment of two packages that sound like they belong here by name but turned out not to both be wired the way their names suggest.

Why it was built

Every signature this codebase produces is meant to be checkable without trusting Parmana's own server to tell you it's valid, that would defeat the purpose of signing anything. Offline verification exists so the check can be performed with zero network calls, zero disk reads beyond the artifact and a public key you already have, and zero environment configuration.

How it works

Offline verification

`verifyExecutionTrustRecordOffline()` (`packages/crypto/src/OfflineVerifier.ts`) is explicitly built to have zero external dependencies at call time: the caller supplies the exact public key material for every `keyId` a record references, and everything else, canonical serialization, the exact field mapping the signature actually covers, the Ed25519/ML-DSA-65 signature implementations, is the same code the real, online `VerificationCrypto` uses. `ExecutionTrustRecordCanonicalView.ts` is shared between the offline and online paths specifically so there is exactly one definition of "which fields participate in the signature," not two implementations that could silently drift apart from each other.

It checks three things, and reports each independently rather than collapsing them into one opaque boolean: `hashValid` (does the record's own `trustRecordHash` match a fresh hash of its canonical content), `legacySignatureValid` (does the always-present legacy `signature` field verify), and `hybridSignaturesValid` (only present when the record actually carries a `signatures` array, a hybrid record needs at least two distinct-algorithm entries, and a duplicate algorithm in that array is treated as invalid). The function's own comment is explicit about a real limitation: the hash check recomputes SHA-256 unconditionally, because that's the only hash algorithm this codebase actually implements today (other config-declared algorithms like `sha3-512` / `blake3` have no real provider), a record hashed under a genuinely different algorithm could not be distinguished from a tampered one by this check alone, since the algorithm used isn't itself recorded on the artifact separately from the hash value.

The comment also names a real relationship worth knowing: this module is the reference a separately published, independently maintained package, `@parmana/sign` (`github.com/pavancharak/parmana-sign`, see `docs/CLAIMS.md` §3.12), is built on, that external package ships the lower-level primitives (canonical serialization, sign/verify) this module also uses, but does not yet know the Execution-Trust-Record-specific canonical field mapping or the hybrid envelope shape. Syncing that into the external package is separate work, not something this repository's own build performs.

Public-key discovery

`createKeysRouter()` (`packages/api/src/routes/keys.ts`) exposes `GET /keys/:keyId` and `GET /.well-known/jwks.json`. Both are deliberately mounted before this app's caller-auth middleware, alongside `/refusal/verify` and `/audit/verify`, a third party cannot be required to already hold a Parmana-issued credential just to fetch the key it needs to check a signature that credential has nothing to do with. Every response includes a PEM-encoded SPKI public key (RFC 7468), which is what `OfflineVerifier.ts` and its Python counterpart both consume directly, plus a `jwk` field when the running Node version's `KeyObject.export({format:"jwk"})` supports the key's algorithm, the comment notes ML-DSA-65 exports as JOSE/COSE key type `"AKP"` (from the IETF draft-ietf-jose-fully-specified-algorithms track, not an identifier this codebase invented) and Ed25519 as the long-standardized `"OKP"`. The JWKS endpoint is explicitly not a standards-pure RFC 7517 JWK Set, since this codebase's PEM-first shape doesn't fit that spec's structure exactly, it's a superset any consumer that only wants the `.jwk` field per entry can filter down to.

An honest assessment of `@parmana/replay` and `@parmana/receipt`

This is worth stating precisely rather than by reputation. Checking actual imports (not package existence, actual usage inside `packages/api/src`):

- **`@parmana/replay`** is a real package (`packages/replay/`), with a real `ReplayEngine`, `ReplayExecutor`, `ReplayBuilder`, `ReplayVerifier`, and a real test suite including an integration test and a determinism test. It has **zero imports anywhere inside `packages/api/src`**. It is real, tested, tutorial-capable code with no production wiring , say this plainly rather than implying it's load-bearing because its name suggests it should be.
- **`@parmana/receipt` , as a standalone package, does not exist in this repository at all** , there is no `packages/receipt/` directory and no `package.json` anywhere named `@parmana/receipt`. Receipt functionality is real and genuinely wired into production, but it lives inside `@parmana/crypto` (`ReceiptCrypto.ts`, `ReceiptHasher.ts`) and the `Receipt` type in `@parmana/shared`, used directly by the runtime's trust-record construction, confirmed by real signed receipts appearing in real `ExecutionTrustRecord` output. If you have read or been told that "`@parmana/receipt` and `@parmana/replay` are both unwired," that claim is now only half true: receipts are real and live; replay is the one that remains genuinely unwired.

How it enables things, with a concrete example

`examples/tutorials/05-verification/run.ts` and `examples/tutorials/12-envelope-verification/run.ts` exercise the online verification path. `packages/crypto/tests/unit/` (search for `OfflineVerifier` or `offline-verifier`) covers the offline path directly with real, generated key material and a deliberately tampered record to confirm detection. `python/parmana/crypto/offline_verifier.py` is the Python-side equivalent of the same capability, for a consumer with no Node.js runtime at all.

How to validate this yourself

- `packages/crypto/src/OfflineVerifier.ts`, `ExecutionTrustRecordCanonicalView.ts`
- `packages/api/src/routes/keys.ts`
- `python/parmana/crypto/offline_verifier.py`
- `packages/replay/src/` and `packages/replay/tests/` (real, but check its own package.json's consumers, or the lack of any inside `packages/api/src`, to confirm the unwired claim yourself)
- `packages/crypto/src/ReceiptCrypto.ts`, `ReceiptHasher.ts` (confirm receipts are real by tracing their usage from `packages/runtime/src/` into an actual trust record)

Integration requirements

None for offline verification itself, that is the entire point, zero network calls, zero environment variables. To exercise `GET /keys/:keyId` or the JWKS endpoint against a real deployment, only `PARMANA_KEY_DIR` (or the configured `KEY_PROVIDER`) needs to be set correctly server-side; the caller needs nothing.

Chapter 19: Health and Readiness

What it is

Two distinct endpoints answer two distinct questions a deployment orchestrator needs answered separately: `GET /health` asks "is this process running at all," and `GET /ready` asks "is this process actually able to serve real traffic right now." Conflating them is a common mistake this codebase deliberately avoids.

Why it was built

A process that is up but backed by dead storage is not actually ready to serve traffic, and a PaaS orchestrator that only checks liveness has no way to route around it. `/ready` exists specifically to catch that case and let the orchestrator act on it.

How it works

`GET /health` (`packages/api/src/routes/health.ts`) is deliberately trivial: it responds `{"status": "UP"}` unconditionally, with no external check of any kind. This is a pure liveness probe, if the process can answer HTTP at all, it answers this.

`GET /ready` (`packages/api/src/routes/ready.ts`) is genuinely different: when storage is not Supabase-backed (`NODE_ENV=test`, or `PARMANA_STORAGE` set to anything other than `"supabase"`), there is no external dependency to probe, so it reports `READY` unconditionally, the same distinction `assertStorageConfigured.ts` already draws at boot. Otherwise, it actually queries Postgres with `SELECT 1` (cheap, transfers no table rows, just confirms the connection and credentials work) via `PostgresPoolFactory`, not `supabase-js`/PostgREST, the route's own comment notes this probe previously depended on PostgREST for a `consumed_nonces` HEAD request, meaning a PostgREST-layer outage could make the readiness probe itself unreliable; querying directly removes that failure mode. A failed query returns `503` with `status: "NOT_READY"` and the real error message as `reason`.

Both responses also carry an `authDisabled` field, and a `warning` string when true. This exists because `PARMANA_AUTH_DISABLED=true` is exactly the kind of misconfiguration ("copied `.env.example` without reading every line") that is easy to miss as a one-time `console.warn` at boot but easy for an operator's own monitoring or synthetic checks to catch and alert on directly when it's a field on an endpoint already polled every 30 seconds.

The incident this route was caught in, the night of 2026-09-16

`/ready`'s own test coverage, `examples/tutorials/89-readiness-probe/run.ts`, runs three scenarios: `NODE_ENV=test` reports `READY` with zero database calls; explicit `PARMANA_STORAGE=memory` outside test mode also reports `READY` with zero database calls; and a `DATABASE_URL` pointed at a genuinely unreachable address (`postgresql://unreachable:unreachable@127.0.0.1:1/postgres`) reports `NOT_READY`, `503`, with a specific reason.

That third scenario broke twice, from two different, real bugs found and fixed the same night. First: `PostgresPoolFactory.create()` (`packages/storage/src/postgres/PostgresPoolFactory.ts`) had no `connectionTimeoutMillis` set, so a connection attempt against a genuinely unreachable address could hang indefinitely instead of failing fast, fixed by adding a 5-second timeout, generous for any real, reachable Postgres instance and short enough that a real outage or this readiness check surfaces promptly. Second, and more subtly: a first version of the same night's `SupabasePolicyRepository` fix (Chapter 7) constructed it, and therefore called `PostgresPoolFactory.create()`, **eagerly at module import time** inside `packages/api/src/application.ts`. Because `PostgresPoolFactory` is a process-wide singleton (`if (this.pool) return this.pool;`), that eager call created and cached a pool connected to whatever `DATABASE_URL` was actually configured at import time, before the tutorial's Scenario 3 ever got a chance to override it to the deliberately-unreachable address. The result: Scenario 3 kept reporting `READY`, using the real, working pool, no matter what `DATABASE_URL` the tutorial set afterward. The fix was to make that construction lazy, the same discipline `packages/api/src/repositories.ts`'s existing `lazyRepository()` helper already establishes for every other repository in this codebase (G-15), and for the identical reason: importing a module must never itself open a live database connection as a side effect; only an actual repository call should. After both fixes, the tutorial's Scenario 3 correctly reports `NOT_READY`, `503`, with `reason: "connect ECONNREFUSED 127.0.0.1:1"`.

This is a genuinely instructive incident, not just a bug fixed in passing: it shows how a singleton connection pool and an eager module-scope side effect can combine to make a test that looks correct on paper silently test nothing at all, and it's exactly the kind of thing a readiness-probe tutorial exists to catch before it reaches production.

How it enables things, with a concrete example

`examples/tutorials/89-readiness-probe/run.ts` is the concrete example, read it directly, it is short and the three scenarios described above are exactly what it runs, printing the real status code and body for each.

How to validate this yourself

- `packages/api/src/routes/health.ts`, `packages/api/src/routes/ready.ts`
- `packages/storage/src/postgres/PostgresPoolFactory.ts` (note the `connectionTimeoutMillis` comment, which names this exact incident)
- `packages/api/src/application.ts` (the lazy `getPolicyRepository()` / `Proxy` construction, and its own doc comment explaining why an earlier, eager version was wrong)
- `packages/api/src/repositories.ts` (`lazyRepository()`, the pattern the fix now mirrors)
- `examples/tutorials/89-readiness-probe/run.ts`
- `packages/storage/tests/unit/postgres-pool-factory.test.ts` (asserts the exact `Pool` constructor arguments, including the timeout)

Integration requirements

No additional configuration beyond what storage already requires (`PARMANA_STORAGE`, `DATABASE_URL` when Supabase-backed). A PaaS orchestrator should be configured to poll `/health` for restart decisions and `/ready` for traffic-routing decisions, treating them as interchangeable defeats the reason both exist.

Chapter 20: Integrating Parmana (Requirements and Getting Started)

What it is

Integrating Parmana means getting a caller, human, AI agent, or automated service, to submit a Business Transaction to a running Parmana instance and receive back a signed Execution Trust Record. There are three real paths to do this: the official TypeScript SDK (`typescript/`, published as `@parmana/sdk`), the official Python SDK (`python/`, published as `parmana`), or plain HTTP against the REST API directly. All three ultimately call the same two routes, `POST /execute` and `POST /transactions`, which run through the identical execution pipeline.

Why it was built this way

A caller hand-building a request body has to keep three id pairs internally consistent (`metadata.businessTransactionId` must equal the top level `businessTransactionId`, `authorization.authorityId` must equal `authority.authorityId`, `intent.authorizationId` must equal `authorization.authorizationId`). Getting any of these wrong produces a 400 from `BusinessTransactionValidator`. The SDKs exist specifically to make that class of mistake structurally impossible, not merely documented. `typescript/src/builders/createBusinessTransaction.ts` says this directly in its own doc comment: this exact mismatch is "the class of mistake that produced every 'X must match Y' 400 response documented in `END-TO-END-FLOW.md`."

How it works

The TypeScript SDK

`@parmana/sdk`'s entry point is `ParmanaClient` (`typescript/src/client/ParmanaClient.ts`). Its own doc comment is explicit about what it does and does not do: it holds configuration, composes sub-APIs, and delegates, it does not evaluate policy, authorize execution, execute business logic, verify trust records, or replay executions, all of that is server side.

Construction requires a `Configuration` (`typescript/src/config/Configuration.ts`):

```
interface Configuration {
  readonly endpoint: string; // Parmana Runtime endpoint, e.g. https://runtime.example.com
  readonly apiKey?: string; // Bearer key minted by scripts/generate-api-key.ts
  readonly timeout?: number; // default 30000ms
  readonly retryPolicy?: RetryPolicy;
  readonly transport?: Transport; // defaults to the SDK's own HTTP transport
  readonly userAgent?: string;
}
```

`apiKey` is optional only against a Runtime started with `PARMANA_AUTH_DISABLED=true`, which is local development only, every other deployment rejects an unauthenticated request with a 401 before a Business Transaction is even constructed.

`ParmanaClient` composes ten sub APIs (`HealthApi`, `ExecutionApi`, `VerificationApi`, `ReplayApi`, `ReceiptApi`, `TransactionApi`, `TrustRecordApi`, `PolicyApi`, `RefusalApi`, `AuditApi`), each a thin wrapper over one or more HTTP calls through the shared `Transport`. The public surface includes `execute()`, `createTransaction()` (a second, independent entry point into the identical pipeline via `POST /transactions`), `verify()`, `getLatestVerification()`, `replay()`, `receipt()`, `transaction()` / `transactions()`, `trustRecord()`, `validatePolicy()`, `refusalRecord()` / `verifyRefusalRecord()`, and `verifyAuditEvent()`.

`createBusinessTransaction()` (`typescript/src/builders/createBusinessTransaction.ts`) is the recommended way to build a request body. It takes the fields a caller actually decides (`principalId`, `purpose`, `action`, `target`, `parameters`, `policy`, `signals`, and optional `businessTransactionId`, `authorityType`, `correlationId`, `tenantId`, `sourceSystem`, `submittedBy`) and derives every id-consistency field automatically. `authorityType` defaults to `"SERVICE"`, described in the source as "the correct value for an autonomous agent" (there is no `"AGENT"` value server side, a common first-integration mistake). `businessTransactionId` defaults to a fresh `crypto.randomUUID()` if omitted, and if you provide your own, you are responsible for its uniqueness, since it doubles as the server's idempotency key.

The Python SDK

`python/parmana/client.py` mirrors the same shape: a `ParmanaClient`, a `ClientConfig` (`python/parmana/config/client_config.py`), and a parallel set of API modules under `python/parmana/api/` (`execution_api.py`, `policy_api.py`, `transaction_api.py`, `trust_record_api.py`, `verification_api.py`, `replay_api.py`, `receipt_api.py`, `refusal_api.py`, `audit_api.py`). `python/parmana/builders.py` is the equivalent of the TypeScript builder. Requires Python 3.10+ (from `pyproject.toml`'s `requires-python`).

The plain HTTP path

Every SDK call ultimately becomes an HTTP request. A minimal integration with no SDK at all needs:

1. A bearer key: `Authorization: Bearer <key>` on every request except `/health`, `/ready`, `/openapi.yaml`, `/openapi.json`, `/api-manifest.json`, `/documentation`, `/reference`, `/refusal/verify`, `/audit/verify`, and the key-discovery routes.
2. A policy that already exists on the target instance, at the exact `(name, version)` you reference, and (if `POLICY_EXECUTION_VERIFICATION_ENFORCED=true` on that instance) has a real `PolicyChangeApprovalRecord` (see Chapter 7).
3. A well formed `BusinessTransaction` body, with every fact your chosen policy's rules reference present in `signals`, matching either a `boundSignals` entry (derived automatically from `target` / `parameters` server side, but still worth sending consistently) or an `unboundSignalReasons` entry (must be supplied directly).

Concrete example

`examples/tutorials/` has dedicated SDK walkthroughs (grep `examples/tutorials/` for `sdk` in the directory name): the TypeScript and Python quickstart, AI-agent, and production-pattern tutorials each demonstrate a real client construction and a real `execute()` call against a locally spun-up server, no live network needed. `docs/site/guides/typescript-sdk-quickstart.mdx` and `docs/site/guides/python-sdk-quickstart.mdx` cover the same ground for the docs site; their code samples were checked against the actual SDK source above, not copied blind.

How to validate this yourself

- `typescript/src/client/ParmanaClient.ts`, `typescript/src/config/Configuration.ts`, `typescript/src/builders/createBusinessTransaction.ts`, `typescript/test/` for the SDK's own test suite.
- `python/parmana/client.py`, `python/parmana/config/client_config.py`, `python/parmana/builders.py`, `python/tests/`.
- `packages/runtime/src/validators/BusinessTransactionValidator.ts` for the exact cross-consistency checks a hand built transaction must satisfy.
- `END-TO-END-FLOW.md` (repo root) for the original incident record of what happens when these checks fail.

Requirements checklist

On the caller side, to make any real call:

- A bearer key, provisioned via `npx tsx scripts/generate-api-key.ts --caller-id "<you>" --credential-holder-type <TYPE>`. `credentialHolderType` must be `USER` if you intend to act as a policy governance checker (Chapter 7); `SERVICE` is typical for an automated integration.
- `allowedPrincipalIds`, if your key is scoped to specific principals, must include whatever `principalId` you submit, or the request is rejected before policy runs. Check with `GET /callers/me`.
- `allowedCapabilities`, if scoped, must include the `action` capability you intend to invoke.

On the server/instance side, verified against `.env.example` and `packages/shared/src/config/Config.ts`:

Variable	Purpose
`PARMANA_POLICY_DIR`	Local policy file directory (used when storage is not Supabase backed).

Variable	Purpose
`PARMANA_KEY_DIR` / `PARMANA_KEY_MATERIAL_JSON`	Local Ed25519 key material (`KEY_PROVIDER=local`).
`PARMANA_STORAGE`	`memory` or `supabase`. `supabase` requires `DATABASE_URL`.
`DATABASE_URL`	Direct Postgres connection string (Supabase or otherwise), used by `pg`, not PostgREST.
`SUPABASE_URL` / `SUPABASE_SERVICE_ROLE_KEY` / `SUPABASE_ANON_KEY`	Only needed for the handful of code paths that still use the Supabase client directly (e.g. CI's read-only policy verification script), not the main request path.
`CRYPTO_MODE`, `PRIMARY_SIGNATURE_PROVIDER`, `SECONDARY_SIGNATURE_PROVIDER`, `HASH_PROVIDER`	Signature/hash algorithm selection (see Chapter 3).
`KEY_PROVIDER`	`local` is the only implemented value as of this writing; anything else throws explicitly (`KeyBootstrap.create()`).
`AWS_REGION`, `AWS_ROLE_ARN`	Required only when using real AWS KMS signing (`KmsSigner.ts`), via Vercel OIDC, no static AWS credentials needed.
`EXECUTION_AUTHORIZATION_TTL_SECONDS`	Default 120.
`PARMANA_API_KEYS`	JSON array of caller entries, the live authentication table.
`RATE_LIMIT_EXECUTE_PER_MINUTE`, `RATE_LIMIT_HEALTH_PER_MINUTE`	Rate limiting (Chapter 16).
`HUBSPOT_PRIVATE_APP_TOKEN`, `GITHUB_APP_ID` / `GITHUB_INSTALLATION_ID` / `GITHUB_APP_PRIVATE_KEY`, `PAYTM_CONNECTOR_URL` / `PAYTM_CONNECTOR_SHARED_SECRET`	Only needed if you intend to exercise the corresponding connector (Chapter 12); a connector is simply not registered if its config is unset, nothing else fails.

Common first-integration mistakes

- Sending `authorityType: "AGENT"`, which does not exist server side; use `"SERVICE"`.
- Hand-building the three id-consistency fields instead of using `createBusinessTransaction()`, and getting one wrong.
- Forgetting that `businessTransactionId` is an idempotency key: resubmitting the same id with different content is rejected as a duplicate before execution, not silently overwritten (see `docs/site/guides/end-to-end-paytm-flow.mdx`).
- Referencing a policy `(name, version)` that exists on disk but has never been approved, on an instance where `POLICY_EXECUTION_VERIFICATION_ENFORCED=true` (Chapter 7). On most current deployments this flag is off, so this specific failure mode is currently latent, not active, but worth knowing about before it's turned on.
- Omitting a `signals` entry for a fact the target policy's rules actually reference; the request is rejected, not silently evaluated with a missing/undefined fact.

Chapter 21: Deployment

What it is

This codebase ships with real configuration for two deployment targets: Vercel (`vercel.json` , serverless Functions) and Fly.io (`fly.toml` , `Dockerfile` , a long-running process). Both run the identical application code; what differs is the execution model, and that difference has produced at least one real production incident worth understanding in detail.

Why it was built this way

Two genuinely different deployment shapes were supported deliberately: Vercel for zero-maintenance serverless hosting with automatic scaling, Fly.io for a conventional always-on process with a persistent filesystem and predictable warm state. Neither is presented as strictly better in this codebase; they trade off differently, and the `PARMANA_STORAGE` /repository-selection logic (Chapter 13) is written to work correctly under both.

How it works

Vercel

`vercel.json` :

```
{
  "buildCommand": "npm run openapi && ./node_modules/.bin/tsc -b",
  "installCommand": "npm install --include=dev",
  "rewrites": [{ "source": "/(.*)", "destination": "/api" }],
  "functions": {
    "api/index.ts": {
      "includeFiles": "{openapi/**,policies/**,node_modules/swagger-ui-dist/**,typescript/package.json,python/pyp
    }
  }
}
```

Every route is rewritten to a single serverless Function, `api/index.ts` . `includeFiles` is notable: it explicitly bundles `policies/**` into the deployed Function, since the Function's filesystem is otherwise whatever the build produced, nothing more. This is also the exact filesystem that turned out to be **read only** at runtime, which is the deployment-specific incident below.

Fly.io

`fly.toml` :

```
app = 'parmana-api'
primary_region = 'bom'

[http_service]
  internal_port = 3000
  auto_stop_machines = false
  min_machines_running = 1

  [[http_service.checks]]
    interval = "30s"
    method = "GET"
    path = "/ready"
```

A conventional container (built from the repo's `Dockerfile`), always running at least one machine, with Fly's own health check hitting `GET /ready` every 30 seconds, the same readiness probe covered in Chapter 19. Unlike Vercel, this filesystem is genuinely persistent and writable for the life of the machine.

The read-only filesystem incident, a concrete illustration

`PolicyChangeApprovalService.approve()` (Chapter 7) writes the newly approved policy content via `PolicyRepository.save()`. The original implementation, `FilePolicyRepository`, writes to the local filesystem, this works perfectly on Fly.io (writable, persistent) and in local development, but Vercel's serverless Functions run on a **read only** filesystem outside `/tmp`. The very first real production approval attempt against the deployed Vercel instance failed immediately with `EROFS`. The fix, `SupabasePolicyRepository` (added 2026-09-16, see Chapter 13), writes to a `policies` table in Postgres instead, which works identically on both platforms. This is the clearest concrete example in this codebase of why "which platform am I deploying to" is not a cosmetic choice, it changes which filesystem assumptions are safe to make in application code.

AWS KMS and Vercel OIDC

For real production signing (as opposed to a local Ed25519 key file), this codebase integrates with AWS KMS via Vercel's native OIDC federation, no static, long-lived AWS credentials stored anywhere. `KmsSigner.ts` reads exactly two environment variables directly:

```
AWS_REGION       # e.g. ap-south-1
AWS_ROLE_ARN     # the IAM role Vercel's OIDC token assumes
```

(Confirmed by grepping `packages/crypto/src/providers/signer/KmsSigner.ts` directly; these are the only two `process.env` reads in that file, `KmsSigner requires AWS_REGION to be set` is a real thrown error if `AWS_REGION` is missing.) `docs/operations/aws-kms-vercel-oidc-setup-guide.md` documents the Vercel side of the setup (`vercel env add AWS_REGION production`, and the trust policy connecting Vercel's OIDC issuer to the IAM role). `docs/operations/2026-09-15-kms-migration-troubleshooting-guide.md` records the real incidents hit while migrating to this from static keys, cross-check its specific claims against current source rather than assuming they still describe live bugs; the migration referenced there is complete as of this session.

`KEY_PROVIDER=local` remains the only fully implemented local-key path; setting `KEY_PROVIDER` to anything else (`aws-kms`, `azure-key-vault`, `gcp-kms`, `hsm` were once silently ignored, a real gap closed by making `KeyBootstrap.create()` throw explicitly naming the unimplemented value, see `docs/VERIFICATION-GAPS.md` gap 40 in the July session's table). The actual KMS signing path used in production goes through `KmsSigner` / `SignerKeyProviderAdapter` independently of that specific `KEY_PROVIDER` flag, verify the current wiring in `packages/crypto/src/SignerBootstrap.ts` if you need the precise selection logic.

Concrete example

There is no example tutorial that actually deploys to either platform (that would require real cloud credentials and is out of scope for a hermetic tutorial suite), but `docs/site/guides/deploy-patterns.mdx` and `docs/operations/aws-kms-vercel-oidc-setup-guide.md` walk through the real, current deployment steps for each target.

How to validate this yourself

- `vercel.json`, `fly.toml`, `Dockerfile` (repo root) for the two real platform configs.
- `packages/crypto/src/providers/signer/KmsSigner.ts` for the exact AWS env vars actually read.
- `packages/policy/src/SupabasePolicyRepository.ts`'s own doc comment for the EROFS incident account.
- `supabase/migrations/20260916060000_add_policies_table.sql` for the schema the fix depends on.
- `docs/operations/2026-09-15-kms-migration-troubleshooting-guide.md` and `docs/operations/aws-kms-vercel-oidc-setup-guide.md` for the full operational history.

Requirements

Vercel deployment:

- `AWS_REGION`, `AWS_ROLE_ARN` (for KMS signing) or local key material via `PARMANA_KEY_MATERIAL_JSON` (for local Ed25519 signing, not recommended for production but functional).

- `PARMANA_STORAGE=supabase` with `DATABASE_URL` set, since the filesystem cannot be relied on for anything written at runtime, including approved policy content.
- Every other env var from Chapter 20's requirements table.

Fly.io deployment:

- The same core env vars, `PARMANA_STORAGE=memory` is viable here if you genuinely want no external database (the filesystem is real and persistent), but `supabase` remains the production-realistic choice for durability across machine restarts.
- `fly.toml`'s health check expects `GET /ready` to respond within its configured timeout (5s); see Chapter 19 for what that route actually checks.

Chapter 22: Testing Philosophy

What it is

Two disciplines run through this codebase's tests: hermetic-first testing (no real network, no real database, no real cloud credentials required to run the suite), and treating the runnable tutorial suite under `examples/tutorials/` as a first-class form of documentation, not a separate, optional thing bolted on after the real code and real tests exist.

Why it was built this way

A test that needs a live Supabase connection or real AWS credentials to run cannot run in CI without secrets, cannot run offline, and is slow. A prose document describing behavior can silently go stale the moment the underlying code changes, since nothing forces it to be re-checked. Both problems have the same root cause, a gap between what is claimed and what is actually exercised, and this codebase addresses both with the same tool: code that asserts real behavior and fails loudly the moment that behavior regresses.

How it works

Hermetic-first testing

Rather than mocking at the framework level, unit tests in this codebase commonly construct a minimal fake object implementing just the one method actually called. `packages/storage/tests/unit/` has several real examples (`business-transaction-repository-duplicate-consistency.test.ts`, `postgres-rate-limit-store.test.ts`, `supabase-business-transaction-repository.test.ts`, among others) that build a `FakePool`-shaped object standing in for `pg.Pool`, exposing only `.query(sql, params)` against an in-memory data structure, no real network socket ever opens. This same pattern is used in the example tutorials that exercise real Postgres-backed classes hermetically (`examples/tutorials/115-per-limiter-rate-limit-stores`, `116-supabase-policy-repository`, `117-maker-checker-one-shot-scripts`, each fakes just enough of `pg.Pool` for the class under test to function correctly, verified by reading their `run.ts` files directly).

`vitest.config.ts` (repo root) sets `NODE_ENV=test` implicitly for the whole suite via Vitest's own default behavior, and `packages/storage/src/StorageFactory.createFromEnvironment()` reads that explicitly, returning `MemoryStorageProvider` unconditionally under `NODE_ENV=test`, regardless of what `PARMANA_STORAGE` happens to be set to in the ambient environment. This exists specifically so importing application code during test collection never accidentally constructs a live Supabase client (see the `G-15` reference in that factory's own doc comment).

The tutorial suite as documentation

`examples/tutorials/` currently runs to 117 numbered tutorials plus `examples/scenarios/`, invoked together via `npm run examples` (`scripts/run-examples.ts`). Each tutorial is a standalone, runnable script that exercises real production code (not a simplified re-implementation) against a hermetic environment, either an in-process Express server on an ephemeral port, or a fake client standing in for one external dependency, and prints its own pass/fail assertion at the end. The reasoning, stated directly in this codebase's own review of itself: a runnable tutorial is falsifiable in a way prose is not, if the underlying behavior changes, the tutorial's own final assertion fails the next time anyone runs it, where a paragraph of documentation making the same claim would simply keep reading as true.

Two entries are deliberately excluded from the automated `npm run examples` run: `examples/04-verified-execution` (binds a real TCP port not safe to run unattended alongside the rest of the list) and `examples/tutorials/09-rest-api` (POSTs to a live server the runner does not start), each documented with a comment in `scripts/run-examples.ts` explaining exactly why, and pointing at that example's own `README.md` for how to run it individually.

Citation integrity

`docs/CLAIMS.md` and the tutorial `README.md` files cite real file paths as evidence for their claims. This codebase treats a stale citation as a real defect worth finding and fixing, not a cosmetic nit, a citation nobody automatically checks will eventually go stale, and a claim whose evidence has silently rotted is functionally the same as an unverified claim. `docs/CLAIMS.md` is explicit that a citation should be understood in the context it was written (a citation may be correct as of the date it was

added even if later superseded, in which case the surrounding prose says so rather than the citation being silently rewritten to point somewhere else, doing so would falsify the historical record rather than fix a real problem).

CI

`.github/workflows/ci.yml` runs, on every push and pull request: install (`npm ci`), build (`npm run build`), lint (`npm run lint`), typecheck (`npm run typecheck`), a guard against retired terminology (a grep-based check, not a type or lint rule), the full test suite (`npm test`), and, in a separate job, `scripts/verify-policy-changes-approved.ts` against whichever `policies/**/policy.json` files changed in that push or PR, this is the fail-closed governance gate described in Chapter 7, real and enforced in CI today, not yet wired as a GitHub required status check due to a plan/visibility limitation external to this codebase (see `docs/CLAIMS.md` §2.26's "Preventive Git-layer enforcement" entry for the exact constraint).

The local pre-commit hook (this repo's own `.git` hooks, not GitHub's CI) additionally runs the full example suite (`npm run examples`) before allowing a commit, meaning every commit landed on `main` has, at minimum, had all currently-registered tutorials pass on the committing machine at commit time.

Concrete example

Any tutorial run demonstrates the philosophy directly:

```
npx tsx examples/tutorials/117-maker-checker-one-shot-scripts/run.ts
```

spins up a real Express app on an ephemeral local port, exercises the real maker-checker HTTP endpoints and the real operational scripts used to approve production policies, and prints a final `✓` / `✗` verdict, entirely hermetically, no network access, no real credentials.

How to validate this yourself

- `packages/storage/tests/unit/` for the fake-`pg.Pool` unit test pattern.
- `scripts/run-examples.ts` for the full registered tutorial list and the two documented exclusions.
- `.github/workflows/ci.yml` for the exact CI gates.
- `docs/CLAIMS.md` 's own header/preface section for the citation-integrity discipline stated in its own words.
- Any individual `examples/tutorials/*/README.md`, each states what it proves and why it matters, in the same structure this book uses.

Requirements

None beyond what's already needed to run the codebase locally (Node.js, `npm install`). No tutorial or unit test in the default `npm test` / `npm run examples` runs requires real AWS, Supabase, or connector credentials; that is the entire point of the hermetic-first discipline described above.

Chapter 23: History and Open Questions

What this chapter is

Every other chapter in this handbook describes what Parmana does today. This one describes how it got there: the real incidents, the corrections, the things that were tried and reversed, and the questions that remain genuinely open rather than quietly resolved. The two files this chapter draws from most, and which a reader should treat as the primary sources rather than this summary, are `docs/VERIFICATION-GAPS.md` (a dated incident and gap log going back to the original external audit) and `docs/CLAIMS.md` (the audit ledger, one entry per claim with cited evidence). This chapter is the narrative connecting them, written by reading both directly rather than trusting either blindly, and by direct, firsthand knowledge of the two most recent sessions, both from 2026-09-15/16.

Why a history chapter exists at all

A codebase that has been audited, found wanting, fixed, and re-audited multiple times has a choice about how to present that history: quietly rewrite the record so it looks like everything was always correct, or keep the record honest, including the mistakes. This codebase has consistently chosen the second option. `docs/VERIFICATION-GAPS.md`'s own structure, gaps organized by the session that found and closed them, with a dedicated "Decision required" section for things nobody has decided yet, exists specifically so a reader can tell the difference between "this was always solid" and "this broke once, here is exactly how, and here is what changed." Chapter 17 of the older `docs/book/` calls this citation integrity; whatever it is called, the discipline is the same: a claim without a file and line citation is not trusted, in this document or in the ones it draws from.

The broad shape of the project's history

Rather than repeating every entry in `docs/VERIFICATION-GAPS.md` (over 2,900 lines as of this writing, covering audits from an original external review through at least seven distinct closure sessions), this section names the eras and points at where to read more:

- **The original external audit (dated July 2026 and earlier).** The root-level `docs/000-CONSTITUTION.md` through `docs/017-CONFORMANCE.md`, plus `ARCHITECTURE.md`, `SPECIFICATION.md`, `TRUST_MODEL.md`, `GUARANTEES.md`, and `PROOFS.md` date from this period. They are frozen, not deleted, and should not be read as current, per `docs/book/`'s own README.
- **2026-07-17 audit closeout session.** Nine tasks closing findings from a July 16 external audit. See `docs/VERIFICATION-GAPS.md`'s "Gaps closed in the 2026-07-17 audit closeout session" section.
- **Phase 3D certification session.** A further, larger closure pass; see that session's own heading in `docs/VERIFICATION-GAPS.md`.
- **2026-09-07 through 2026-09-14: a run of focused sessions** covering production readiness, real-deployment verification, PQC production-readiness remediation, and execution-audit-trail hardening, each with its own dated section in `docs/VERIFICATION-GAPS.md`.
- **Gaps 35 through 40 (audit-artifact-driven, same day as the gaps-24-34 pass).** These closed detection-then-prevention gaps in Policy Governance specifically: a five-minute periodic integrity re-check (gap 36), a legacy-policy backfill script (gap 39), and execution-time prevention itself (gap 40, `PolicyExecutionVerifier`, feature-flagged off by design at the time it was built, since every real production policy was still `PENDING_APPROVAL`). Chapter 7 of this handbook covers what changed about that precondition on 2026-09-16.
- **2026-09-15/16: the AWS KMS migration and Policy Governance completion.** The two most recent sessions, covered in detail below, since they are the freshest and least likely to already be reflected accurately anywhere else.

The 2026-09-15 AWS KMS gateway signing migration

The gateway's signing key moved from a local file-based key to AWS KMS, authenticated via Vercel OIDC rather than static AWS credentials. Six real bugs were found and fixed in the course of this migration; `docs/operations/2026-09-15-kms-migration-troubleshooting-guide.md` is the detailed account. The one worth naming specifically here, because of how it was found: `SignerKeyProviderAdapter` (packages/crypto) had a divergence between the key used to *sign* and the key used to *verify*, a signing/verification key mismatch that would have silently produced signatures nothing could verify, had it

reached production unnoticed. It is documented as G-48/G-49 in `docs/VERIFICATION-GAPS.md`, and Chapter 3 of this handbook covers the cryptography layer this fix lives in.

Two other findings from the same session are directly relevant to chapters elsewhere in this handbook and are documented as their own tutorials rather than just prose: `examples/tutorials/113-kms-key-id-resolution` (AWS KMS rejects a bare logical key ID like `"default"`; a resolver maps it to the alias/ARN/key-ID shape KMS actually accepts) and `examples/tutorials/115-per-limiter-rate-limit-stores` (a shared rate-limit `Store` instance reused across two limiters, which `express-rate-limit`'s own contract disallows, including a documented correction to an initially wrong diagnosis: the observed 500 error was blamed on the store-reuse warning by proximity in a log, when the real cause was the separate signing/verification key divergence above; the correction is kept visible rather than silently rewritten, matching this codebase's own stated discipline).

The night of 2026-09-16: closing the Policy Governance backfill

All ten real production policies had sat `PENDING_APPROVAL` since 2026-08-19, waiting on a genuinely distinct second human checker (`SameActorCannotApproveOwnChangeError` meant the original proposer could never approve their own proposals). This session provisioned that checker identity (`policy-reviewer-1`) and approved all ten, plus four more pre-existing policies that had never been proposed through the governance system at all. Chapter 7 covers the mechanism; this section covers what went wrong along the way, since every one of these was a genuine, previously-unexercised bug, not a repeat of something already known.

1. **`EROFS` on the very first live approval.** `PolicyChangeApprovalService.approve()` writes the newly-approved policy content via `PolicyRepository.save()`. `FilePolicyRepository` does that with a local filesystem write. Vercel's serverless Functions run on a read-only filesystem. Because no real approval had ever been attempted against a live, deployed instance before this session (every prior approval attempt in this codebase's history was against local dev or test environments), this code path had simply never yet executed for real. The first time it did, in production, it failed immediately. Fixed with a new `SupabasePolicyRepository` (`packages/policy/src/SupabasePolicyRepository.ts`, migration `20260916060000_add_policies_table.sql`), used whenever a real database is configured.
2. **An eager-construction bug in the first version of that fix.** The first attempt at wiring `SupabasePolicyRepository` into `packages/api/src/application.ts` constructed it, and the `PostgresPoolFactory` singleton pool underneath it, at module import time rather than on first actual use. This violated a discipline already established elsewhere in this exact codebase (`packages/api/src/repositories.ts`'s `lazyRepository`, closing gap G-15, whose own comment states the same rule: importing a module must never itself construct a live database connection). The concrete symptom: constructing the pool eagerly meant it connected using whatever `DATABASE_URL` was live at import time, before `examples/tutorials/89-readiness-probe`'s third scenario (a deliberately unreachable database) could ever configure it differently, so that test's `NOT_READY` assertion silently became `READY`. Fixed by making the construction lazy, mirroring `repositories.ts`'s own pattern exactly.
3. **`PostgresPoolFactory` hanging indefinitely against an unreachable database**, discovered via the same tutorial, instead of failing fast. Fixed by adding `connectionTimeoutMillis: 5000` to the pool configuration.
4. **Stale content in the original 2026-08-19 proposals.** `pending_policy_changes` stores a one-time snapshot of proposed content, taken when the proposal is created, never re-checked against the live file while the proposal sits open. A month is an unusually long time for a proposal to remain open, and in that time the live policy files gained a documentation field (`unboundSignalReasons`, harmless) and, for two policies specifically (`connector-capability`, `customer-refund`), a functional field (`boundSignals`, which `SignalIntentBinder` needs to derive `paymentAmount` / `refundAmount` from a request's own parameters rather than treating them as needing independent verification). Approving the stale content as originally proposed would have quietly persisted that gap. Caught during review, not by any automated check, and fixed by re-proposing and re-approving all fourteen policies with their current, correct content, see Chapter 7 for the verification step (`scripts/verify-policy-changes-approved.ts --full-scan`) that confirms the fix.

None of these four were regressions of something previously working. Each was code, or a data path, that had genuinely never been exercised against a real deployment before this session, caught the first time it actually ran.

Genuinely open questions

These are unresolved by design, not oversights. Each is documented in full, with real options weighed, in `docs/VERIFICATION-GAPS.md`'s "Decision required" section; this is a short index, not a replacement for reading the original

entries.

- **D-2 (partially resolved).** Hybrid/post-quantum signatures are wired for Trust Records and Receipts, opt-in via `CRYPTO_MODE`, but not for `RuntimeAuthorizationSigner`, gateway attestation signing, or connector-signing call sites. Deliberately deferred to a fast-follow milestone, not a rejection.
- **D-3.** `OverrideService` exists, is presumably complete, but is not wired to any real HTTP route and is not exercised by tests beyond a storage-layer bypass. Undecided whether to wire it in or mark it explicitly future-scope.
- **D-4.** HubSpot's `TRUSTED_APPROVAL_ISSUERS` is an empty array by design (fail-closed), meaning every `preAuthorizedForAmountChange` claim currently fails closed since no real approver key has been provisioned. A proposed hardcoded dev key was explicitly rejected during review as inconsistent with this codebase's own never-commit-a-private-key convention. Status: undecided.
- **D-5.** Upstream authorization verification (`NF-001-UPSTREAM-AUTHORIZATION-VERIFICATION.md`) is explicit future scope: today's authorization boundary is caller identity plus principal/capability scoping, not an independent signature on the `Authority` / `Authorization` domain objects themselves. This becomes a real gap only for a delegation scenario (multi-party approval, an external OAuth/SAML/risk-service authorization source), and work has deliberately not started pending a real trigger.
- **D-6.** The CI job that would prevent an unapproved policy file from merging (`scripts/verify-policy-changes-approved.ts`, run by `.github/workflows/ci.yml`'s `verify-policy-approvals` job) is real and fail-closed, but is not configured as a *required* GitHub branch-protection check. Attempting to enable it during a real session failed with `403 Upgrade to GitHub Pro or make this repository public`, an externally imposed constraint (private-repository plan limits), not a gap in this codebase's own logic. See `docs/CLAIMS.md` §2.26's "Preventive Git-layer enforcement" entry for the full account.
- **The internal-vs-external policy authoring question.** Should Parmana be the system of record for policy approval at all, or should policies be authored and approved in an external system, with Parmana staying strictly read-only/enforcement-only? Nothing in the codebase picks a side; the maker-checker system described in Chapter 7 exists because policy authoring was previously outside any governance surface at all, not because building it internally was compared against and preferred over the external alternative.
- **`POLICY_EXECUTION_VERIFICATION_ENFORCED` remains `false`.** As of this session, every real production policy has a genuine, verified approval record, closing the precondition that originally justified leaving this flag off. The flag itself was deliberately left unchanged as part of closing that precondition; turning it on is reserved as its own, separate, later decision (`docs/operations/policy-approval-runbook.md` Part 5), not something to do as a side effect of finishing the backfill.

How to keep this chapter honest going forward

The same rule that produced this chapter applies to updating it: a new incident gets added here (or to `docs/VERIFICATION-GAPS.md`, whichever is the more precise home for it) with a real file citation and an honest account of what broke and why, not a summary that makes the codebase look more finished than it is. If a claim in this chapter is ever found to be wrong against the actual current source, the correction belongs here, visible, the same way the rate-limiter/signing-key misdiagnosis in the 2026-09-15 session is kept visible in `examples/tutorials/115-per-limiter-rate-limit-stores/README.md` rather than quietly rewritten.