

POO²L Platform Complete Guide

POO²L Platform Complete Guide

The headless application core over the POU²L engine

InZeMobile

TABLE DES MATIÈRES

Partie 1 — Foundations

The Platform at a Glance	7
Getting Started	12
Core Concepts: Tenants, Templates, Containers	16

Partie 2 — The Template API

Template Lifecycle and Versioning	23
Containers and Organization	27
Rendering	31
Validation, Analysis and Dependencies.....	35

Partie 3 — Data and Registry Services

The Registry and the Cache.....	41
Enumeration, Events and Blobs.....	44
Packages: Export and Import.....	47

Partie 4 — Storage

The Storage SPI and Shipped Adapters.....	53
Host-Provided Storage.....	57

Partie 5 — Integration Surfaces

The C ABI and Language SDKs.....	63
The HTTP Server	66

Appendices

Complete Method Index.....	71
Data Model Reference	75
Error Reference	80
Guarantees, Gates and Proofs.....	83
Resources.....	87

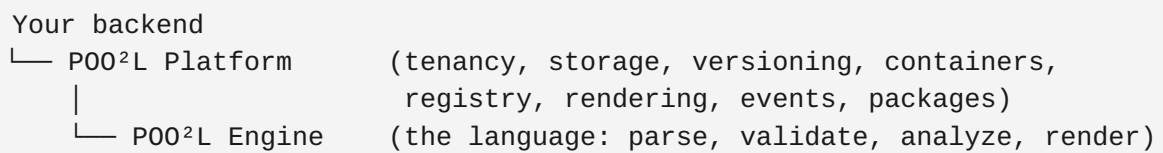
Partie 1

Foundations

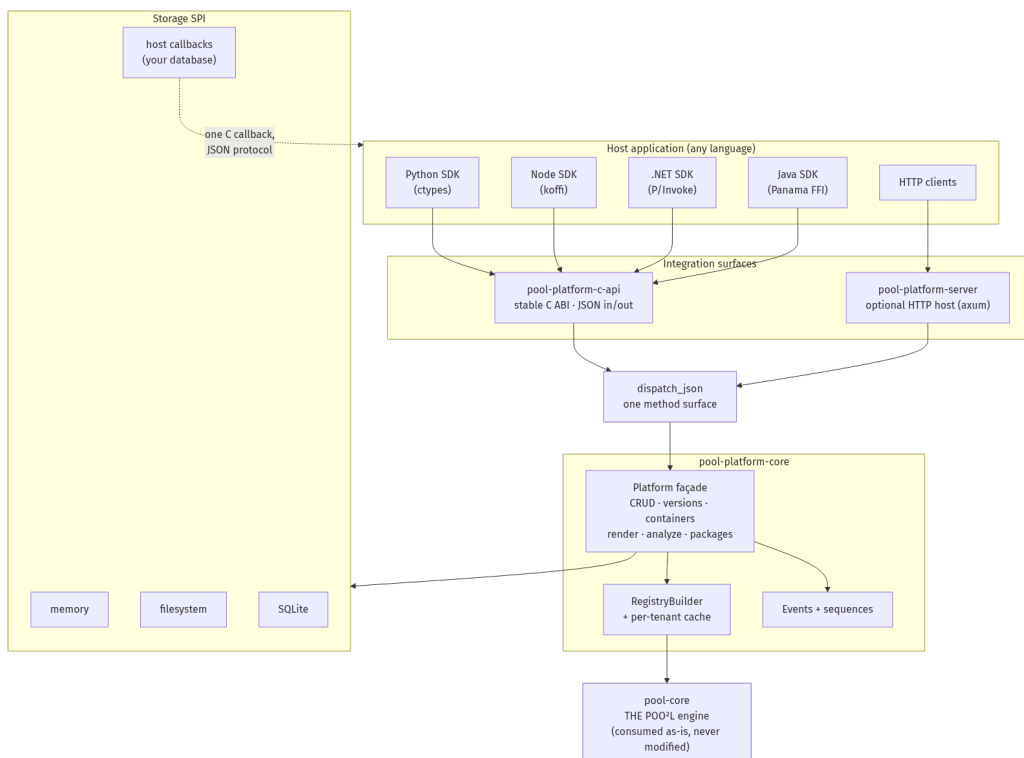
The Platform at a Glance

Chapter I — The Platform at a Glance

The POO²L Platform is a headless application core : everything a backend needs to manage, store, version, resolve and render POO²L prompt libraries, packaged as an embeddable library with no user interface, no HTTP server requirement, and no opinion about your product. It sits exactly between the POO²L engine — the single reference implementation of the language — and whatever application you are building on top :



The project motto captures the division of labor: *the language defines, the engine executes, the Platform industrializes* — and the application (IziBuddy or yours) provides the user experience.



1.1 — Who this book is for

A professional developer who wants to embed POO²L prompt libraries in a backend — a SaaS product, an internal tool, a CLI, an agent framework — without reimplementing storage, versioning, identifier resolution or the render pipeline. The book assumes fluency with JSON and general backend concepts, and no prior knowledge of the Platform codebase. Familiarity with the POO²L language helps for the rendering chapters; where language semantics matter they are summarized here, and the *POO²L Language Reference* holds the full story.

Everything documented in this guide is derived from the Platform's source code (`src/platform/` in the repository) and verified by its test suite. Where the Platform deliberately deviates from a historical behavior, the deviation is called out explicitly.

1.2 — What the Platform is

One sentence per pillar :

- **One engine, consumed as-is.** The Platform embeds `pool-core`, the reference POO²L engine, as a plain Rust dependency. It never reimplements a byte of grammar, and it never modifies the engine (a governance rule enforced by CI — see appendix D).
- **Tenancy is first-class.** Every call takes a tenant identifier; data of different tenants never mixes, in any adapter.
- **Storage is a service-provider interface.** Five small traits (templates, containers, versions, blobs, sequences) plus authorization and event hooks. Three adapters ship in the box — memory, filesystem, SQLite — and a host can implement the traits in its own language through a single C callback.
- **Typed errors, engine diagnostics verbatim.** One error enum crosses every binding; when the engine raises a blocking diagnostic, its SCREAMING_SNAKE code (`UNDEFINED_REFERENCE`, `AMBIGUOUS_INCLUDE`, ...) arrives untouched.
- **Data, never opinions.** The Platform provides enumeration primitives, mutation events and a lossless scan so a host can build search, export and indexing *its* way. It ships no search engine, no ranking, no scoring, no tag taxonomy. The one exchange format it does speak — the standard's §7.4 package — is normative, not application-specific.

1.3 — What the Platform is not

The explicit non-goals, straight from the public contract: tags, scoring, search matching and ranking, proprietary export envelopes, HTTP transports as a requirement (the bundled server is optional), authentication of end users, UI preferences. These are application concerns. The `metadata` bag (chapter 3) exists precisely so a host can attach its own fields to Platform entities without the Platform ever interpreting them.

1.4 — The codebase in one table

The Platform lives in its own Cargo workspace, deliberately a *sibling* of the engine workspace rather than a member of it — joining would require editing a file under the engine tree, which governance forbids.

Where	What
<code>crates/pool-platform-core</code>	The application core : models, façade, registry construction, rendering, analysis, versioning rules, containers, tenancy, cache, events, packages. Depends only on <code>pool-core</code> , <code>serde</code> , <code>uuid</code> , <code>sha2</code> , <code>thiserror</code> .
<code>crates/pool-platform-storage</code>	The filesystem and SQLite adapters (the memory adapter lives in core). SQLite is bundled — no system dependency.
<code>crates/pool-platform-c-api</code>	The stable C ABI : a handful of exported functions, JSON in, JSON out. Also home of the host-callback storage channel. Built as <code>cdylib</code> , <code>staticlib</code> and <code>rlib</code> .
<code>crates/pool-platform-server</code>	The optional HTTP host (axum) : one generic endpoint over the exact same dispatch as the C ABI, plus a Dockerfile.
<code>sdk/</code>	Python, Node, .NET and Java wrappers — thin and logic-free — plus packaging and runnable examples.
<code>benchmarks/</code>	The non-regression benchmark against the historical Python render path.
<code>GOVERNANCE.md</code> , <code>PLATFORM-API.md</code> , <code>ROADMAP.md</code>	Scope freeze, public contract, forward plan.

1.5 — One dispatch, many surfaces

A design decision worth internalizing early, because it explains why this book can document *one* method surface and have it apply everywhere : the C ABI and the HTTP server are two hosts of the **same** dispatch function. A method like `render_template` exists once, in Rust, on the `Platform` façade; `dispatch_json` maps a method name plus a JSON arguments object onto that façade; the C ABI exposes `dispatch_json` behind `poo2l_platform_call`, and the server exposes it behind `POST /v1/{tenant}/{method}`. Every language SDK is a wrapper over the C ABI that contains *no logic* — the Python method `render_template(key, request)` literally sends `{"key": ..., "request": ...}` to the same dispatcher the server uses.

Consequences : the method index in appendix A is valid for Rust, C, Python, Node, .NET, Java and HTTP alike ; error shapes are identical everywhere ; and a behavior verified in the Rust test suite is the behavior every binding gets.

1.6 — The guarantees, up front

The Platform’s test suite is not an afterthought; several of its properties are the product’s acceptance criteria, so they deserve to be stated in chapter 1 :

- **Conformance parity.** The full POO²L standard-conformance pack — 511 cases — is replayed *through the Platform’s render path* (storage → registry construction → en-

gine) and compared byte-for-byte against the pack's expected outputs. Zero divergence is a permanent CI gate.

- **Engine immutability.** No tracked file under the engine tree may change during Platform work; CI fails otherwise.
- **Lossless reconstruction.** `scan` + `list_containers` + `list_versions` + `get_blob` reconstruct a tenant without loss, including opaque metadata bags — the contract behind any host-built export (chapter 9), and the foundation of the native package format (chapter 10).
- **Exactly-one event per mutation.** Every successful mutation emits one event after commit, with a per-tenant strictly increasing sequence number — durable across restarts on the filesystem and SQLite adapters.
- **Performance.** The reference benchmark renders the same corpus through the historical Python path and through the Platform's C ABI, asserts byte-identical output, and requires the Platform to be equal or better.

Appendix D details how each guarantee is enforced.

1.7 — How this book is organized

Part 1 covers the concepts every chapter builds on: tenants, the prompt model with its two identities, containers and the metadata bag. Part 2 is the working API — lifecycle and versioning, containers, rendering, and the static-analysis services. Part 3 documents the machinery that supports data services: registry construction and caching, enumeration and events, and the native package format. Part 4 is for integrators who bring their own persistence: the storage SPI, the shipped adapters, and host-provided storage through C callbacks. Part 5 covers the integration surfaces themselves — the C ABI, the four SDKs, and the HTTP server. The appendices collect the flat method index, the data-model reference, the error reference, and the guarantees with their proofs.

Templates and skills

A stored `content` is the authoritative POO²L source. Declare `@@@template` + `@@@type skill` to create or convert a skill; updating the same UID preserves identity. `analyze` and `render` responses expose `document_type`; list summaries derive it from the source. Older sources remain templates. No independent type property overrides the source. The Python, Node, Java and .NET wrappers return the additive JSON fields unchanged.

For native built-in storage, pass `"mixed_inheritance":"deny"` (default `allow`) in construction config. The HTTP host accepts `P002L_MIXED_INHERITANCE=deny` or `--mixed-inheritance deny`. This is trusted host configuration, never a request field. Rust hosts set `PlatformConfig.mixed_inheritance`, including hosted storage. The C ABI adds `poo2l_platform_new_hosted_policy` for callback storage; Python accepts the same config with `host_storage=`. A denied mixed edge returns `MIXED_DOCUMENT_INHERITANCE_FORBIDDEN`; allowed edges produce warnings. Source validation alone does not resolve inheritance.

User-local drafts belong to IziBuddy's browser, not to this headless platform. A platform host must provide its own private working-state/publication workflow.

Templates and document nature

A **template** is the common reusable entity. Its **Nature** is `prompt`, `skill` or a custom identifier such as `mod` or `acme.review`. Create **New prompt** or **New skill**, then edit Nature in Properties when needed. Changing nature preserves the UUID, references, parameters and history. The leaf template owns its type; includes never change it. Mixed inheritance can be allowed with warnings or denied.

```
@@@template review_assistant
@@@type skill
@@@description Reusable review instructions
@@@section rules
Separate observations from assumptions.
```

Use `type:acme.review` to search a custom nature. The **Example** flag replaces the old Template flag; `flag:example` finds these starting examples. `flag:draft` continues to find private drafts. A prompt copies the rendered text, a skill exports SKILL.md and resources, and other types export composed text without claiming a specialized package format.

Getting Started

Chapter II — Getting Started

This chapter takes you from nothing to a rendered prompt on each of the three integration styles: embedded Rust, an SDK over the C ABI (Python as the example), and the HTTP server. All three routes exercise the same façade, so the concepts transfer verbatim; only the plumbing differs.

2.1 — Building the native artifacts

The Platform workspace lives at `src/platform/`. Two build commands cover every use in this chapter:

```
cd src/platform

# The library crates (core + storage) – enough for embedded Rust:
cargo build --release

# The C ABI shared library – required by every SDK:
cargo build --release -p pool-platform-c-api
# → target/release/libpool_platform_c_api.so    (Linux)
#   target/release/libpool_platform_c_api.dylib (macOS)
#   target/release/pool_platform_c_api.dll      (Windows)
```

The SDKs locate the shared library through, in order: the `P002L_PLATFORM_LIB` environment variable, a bundled native directory next to the SDK (`_native/` in Python, `native/` in Node — populated by the packaging script `sdk/package.sh`), then the workspace build tree (`target/release`, then `target/debug`). During development, building the C ABI in release mode is therefore all it takes. One caveat: because release is probed *before* debug, a stale release library can shadow a freshly built debug one — rebuild release after pulling engine or Platform changes.

2.2 — Route 1 — embedded Rust

The zero-configuration constructor gives a fully working in-memory Platform: memory storage, allow-all authorization, no-op events.

```
use pool_platform_core::{Platform, TenantId, CreateTemplate, RenderRequest};

let platform = Platform::in_memory();
let t = TenantId::new("acme")?;

platform.create_template(&t, CreateTemplate {
```

```

        name: "Code review".into(),
        content: "@@@name Code review\n@@@param language : text = \"python\\\"\\n\\nReview th
        ..Default::default()
    })?;

let out = platform.render_template(&t, "CODE_REVIEW", &RenderRequest {
    parameters: [("language".into(), "rust".into())].into(),
    system: [("USER".into(), "alice".into())].into(),
    ..Default::default()
})?;
println!("{}", out.text);
// Review the following rust code for alice.

```

Three things happened implicitly. The display name « Code review » was derived into the canonical logical id `CODE_REVIEW` (chapter 4 gives the derivation rules). The create snapshotted version `0.1` (chapter 4 again). And the render request's `USER` entry was routed to the engine's *system* context while `language` went to *parameters* — the host computes system-variable values; the Platform only routes them (chapter 6).

For persistence, swap the config:

```

// Filesystem: one directory per tenant, JSON documents.
let platform = Platform::new(pool_platform_storage::filesystem_config("/data/poo2l"))

// SQLite: one database file, bundled libsqlite3.
let platform = Platform::new(pool_platform_storage::sqlite_config("/data/poo2l.db"))

```

2.3 — Route 2 — an SDK over the C ABI

The Python SDK (`sdk/python/`) is a pure-Python ctypes wrapper; no compilation step on the Python side.

```

# PYTHONPATH=src/platform/sdk/python (development), or install the wheel.
from poo2l_platform import Poo2lPlatform

platform = Poo2lPlatform({"storage": "sqlite", "path": "poo2l.db"}, tenant="acme")

rec = platform.create_template({
    "name": "Code review",
    "content": "@@@name Code review\n@@@param language : text = \"python\\\"\\n\\nReview
})
out = platform.render_template("CODE_REVIEW", {
    "parameters": {"language": "rust"},
    "system": {"USER": "alice"},
})
print(out["text"])

```

Every SDK method is a JSON round-trip to the same dispatch: `create_template` sends the arguments object, gets back `{"ok": <TemplateRecord>}` or raises `PlatformError` with the typed error object. The generic escape hatch `platform.call(method, args, tenant=...)` reaches any method in appendix A, including ones added after the SDK was written.

The Node, .NET and Java wrappers follow the same pattern (chapter 13); the runnable end-to-end example is `sdk/examples/demo_backend.py`, which provisions a small library (collections, inheritance, includes, parameters), renders, versions, analyzes, enumerates and stores blobs — asserting every result.

2.4 — Route 3 — the HTTP server

For environments that cannot load a native library, or that want process isolation :

```
cd src/platform
cargo run --release -p pool-platform-server -- --storage sqlite --path /data/poo2l.db
# poo2l-platform-server listening on http://127.0.0.1:7433
```

The whole surface is one generic route — `POST /v1/{tenant}/{method}` with the arguments object as the JSON body — plus `GET /health` :

```
curl -s http://127.0.0.1:7433/v1/acme/create_template \
-H 'content-type: application/json' \
-d '{"name": "Hello", "content": "@@@name Hello\n\nHi @@@=USER"}'

curl -s http://127.0.0.1:7433/v1/acme/render_template \
-H 'content-type: application/json' \
-d '{"key": "HELLO", "request": {"system": {"USER": "alice"}}}'
# {"ok":{"text":"Hi alice","warnings":[],"attachments":{...}}}
```

Errors come back with a mapped status code (404 for `not_found`, 409 for `conflict`, 422 for `engine_diagnostic`, ...) and the same typed error object in the body. Authentication is deliberately not built in — the host owns users and sessions; put a reverse proxy in front or use embedded mode. Chapter 14 covers configuration and the Docker image.

2.5 — Bringing your own storage

All three routes above used shipped storage. The fourth option — the one that makes the Platform fit an existing product — is host-provided storage: implement the storage operations in *your* language, against *your* database, and hand them to the Platform as a callback. In Python :

```
from poo2l_platform import Poo2lPlatform, HostStorage

class MyDbStorage(HostStorage):
    def template_get(self, tenant, args): ...    # your SQL here
```

```

def template_put(self, tenant, args): ...
def template_scan(self, tenant, args): ...
# containers / versions / blobs likewise

platform = Poo2lPlatform(host_storage=MyDbStorage())

```

The Platform then routes every read and write through your code while keeping all the semantics — versioning rules, registry construction, rendering — on its side. Chapter 12 documents the protocol; `sdk/examples/host_storage_demo.py` is the runnable proof.

2.6 — The one-page mental model

Before diving into the reference chapters, the whole Platform fits in six lines :

1. **Tenant**: every call is tenant-scoped ("default" if you don't care).
2. **Template**: `uid` is the storage identity; `template_id` is the logical POO²L id (UPPERCASE, unique per collection). `content` is the POO²L source and the single source of truth. `metadata` is your opaque extension bag.
3. **Containers**: slash paths. `folder` and `project` organize (transparent to addressing); `collection` qualifies member ids (`Coll.MEMBER`); `library` packages a namespace.
4. **Render**: you compute system-variable values (DATE, USER, ...); the Platform builds the engine registry from storage and calls the engine. Warnings come back as data ; blocking diagnostics as typed errors.
5. **Versions**: every content change appends `x.y` ; majors need a note ; identical content never bumps.
6. **Enumeration and events**: field-selected, cursor-paged listing plus a lossless scan, and exactly one sequenced event per mutation — the primitives for *your* search, export and indexing.

Core Concepts : Tenants, Templates, Containers

Chapter III — Core Concepts : Tenants, Templates, Containers

Four concepts carry the whole API : the tenant that scopes every call, the stored prompt with its two identities, the container taxonomy that organizes templates, and the opaque metadata bag that keeps application concerns out of the Platform without forcing hosts into a second store. This chapter defines each one precisely ; the chapters that follow assume them.

3.1 — Tenants

Every Platform call takes a `TenantId` — a non-empty, host-defined string : a username, a customer id, a workspace slug, anything. Data of different tenants never mixes, in any adapter : the filesystem adapter maps each tenant to its own directory, the SQLite adapter carries a `tenant` column in every table, and the in-memory adapter keys everything by tenant first.

Construction validates the id : it must be non-empty, and — because filesystem adapters map tenants to directories — it must not contain path separators (`/`, `\`, `NUL`) nor be `.` or `..`. Hosts that do not care about multi-tenancy pass the default tenant, the literal string `"default"`.

There is no « create tenant » call : a tenant exists as soon as something is written under it. Tenant state on the Platform side (the event sequence counter and the registry cache, chapter 8) materializes lazily on first access.

3.2 — The stored prompt

The Platform's prompt is deliberately smaller than a typical application's. Application fields — tags, colors, UI flags, scoring — live either in the host's own store, keyed by the Platform's `uid`, or inside the `metadata` bag described below. What remains is exactly what storage, resolution and rendering need :

Field	Type	Notes
<code>uid</code>	string	Storage identity — a UUID assigned by the Platform on create. Never the logical id.
<code>template_id</code>	string	Logical POO²L identifier — canonicalized UPPER CASE, unique per (tenant, collection) .

Field	Type	Notes
name	string	Display name.
content	string	The POO ² L source — the single source of truth, inline declarations included.
description	string	Free text.
collection	string or null	Slash-separated container path ("Team/Reviews"), or null for root.
inherits	string or null	Parent prompt's logical id (OO inheritance).
virtual	bool	Abstract prompt : instantiating it directly is an engine error.
disabled	bool	Refuses to render (chapter 6).
deleted / deleted_at	bool / RFC3339 or null	Soft delete — the trash (chapter 4).
aliases	array of strings	Alternative identifiers accepted wherever a key is resolved.
deprecated	string or null	Deprecation message.
diamond_policy	default merge refuse	Diamond-include policy.
created / modified	RFC3339 strings	Set by the Platform (UTC, second precision).
metadata	JSON value	Opaque host extension bag — stored, enumerated, exported, never interpreted.

The two identities

The distinction between `uid` and `template_id` shapes the whole API, so it deserves emphasis.

The **uid** is what storage knows : a UUID minted at creation, stable for the lifetime of the record, unique per tenant. Mutating calls (`update_template` , `delete_template` , `purge_template` , version queries) address templates by uid, because mutation must be unambiguous.

The **template_id** is what the POO²L language knows : the identifier used in `@@@include` and `inherits` references. It is canonicalized on write — trimmed and uppercased, since the engine treats identifiers case-insensitively and storage keeps one canonical form. Uniqueness is scoped to (tenant, collection) : two collections may each contain a `WELCOME` , and qualified references (`Marketing.WELCOME`) tell them apart. Renaming a `template_id` is an ordinary update.

Read-oriented calls (`get_template`, `render_template`, `validate_template`, `analyze_template`, ...) take a **key** and resolve it liberally, in this order: exact uid match first (live templates only); then a case-insensitive scan for a matching `template_id`; then aliases — first match in enumeration order wins. This is a convenience for hosts and humans; software that already holds a uid should pass the uid.

Content as the single source of truth

The `content` field is a complete POO²L source. Anything the language can express — `@@@name`, `@@@description`, parameter declarations, sections, includes — lives *in the content*, and the engine is the only interpreter of it. The structured fields around it (`name`, `description`, `collection`, `inherits`, `virtual`) exist for storage and enumeration; at render time the Platform projects them *into* the source's `@@@template` header so the engine sees one coherent world (chapter 8 explains this projection in detail). One practical consequence: an existing `@@@template` line inside stored content is overwritten by the model's fields on the way to the engine — the record's fields win; only a leading `virtual/final` modifier survives.

3.3 — Containers

Containers are metadata rows keyed by slash path, per tenant — `"Marketing"`, `"Team/Reviews"`, `"LIB/Sub"`. A container row has a `kind`, a `description`, an optional `label`, an optional `inherits` (collections only), a `requires` list (libraries only), and its own opaque `metadata` bag.

The four kinds split into two families:

- **Transparent to addressing** — `folder` and `project`. They organize the tree but do not appear in POO²L identifiers: a prompt stored under folder path `Sorting/Sub` is still just `P` to the language. Projects additionally bound include resolution (they emit scope-boundary markers in the engine registry — chapter 8).
- **Qualifying** — `collection` and `library`. They own a namespace: a prompt `WELCOME` in collection `Marketing` is `Marketing.WELCOME` to the language. Collections may inherit from other collections (the OO² axis); libraries are packaging units with a strict name grammar and `requires` dependencies.

A crucial subtlety: a path segment qualifies **only if a container row for that path exists and is declared** with a qualifying kind. An undeclared path — a prompt whose `collection` field names a path with no container row — behaves as a folder: transparent. This is what makes it safe to organize templates into directories without touching the language's addressing, and to *promote* a folder into a collection later by declaring the row.

Kinds also constrain nesting (enforced on `put_container`, chapter 5): collections nest anywhere; folders and projects may not live inside a collection or a library; libraries may not live inside a collection or another library.

3.4 — Versions

Every prompt carries an append-only version history, maintained automatically by the Platform (never by the host):

```
{ "version": "0.2", "operation": "update", "datetime": "2026-07-12T22:01:37Z",  
  "note": null, "checksum": "a1b2c3d4" }
```

The rules, enforced in one place and identical for every storage adapter: a first save records `0.1`; a content change appends a minor bump (`y+1`); a *major* bump (`x+1.0`) must be explicitly requested and **requires a note**; saving identical content appends nothing (identity is a checksum: SHA-256 of the content with line endings normalized, first 8 hex chars). The `operation` discriminant records why the entry exists: `create`, `update`, `restore` (a rollback re-save), or `migrated`. Chapter 4 walks the full lifecycle.

3.5 — Blobs

Attachments are stored as content-addressed blobs: `put_blob(bytes)` returns a `BlobRef` of the form `sha256:<hex>`, and putting the same bytes twice returns the same ref (adapters refcount, so a release decrements and deletes only when unreferenced). The Platform stores bytes and returns bytes; base64 appears only in JSON transport (C ABI and HTTP), where binary cannot travel natively. The engine's attachment machinery (`@@@attachment` declarations) surfaces at render time as data — chapter 6.

3.6 — The metadata bag

Both templates and containers carry a `metadata` field: an arbitrary JSON value that the Platform stores, enumerates (on request), exports and imports — and **never interprets**. This is the escape hatch that keeps application fields out of the Platform without forcing hosts that want a single store to run one anyway.

By convention — mirroring the POO²L standard's package rule (§7.4) — the bag is namespaced per implementation:

```
{ "x_izibuddy": { "tags": ["review", "team"], "color": "#6eb043" },  
  "x_other_tool": { "pinned": true } }
```

A host owns the block bearing its name and must not modify a foreign block; deleting a foreign block is reserved to an explicit user action. Two API affordances support this discipline:

- The wholesale `metadata` replace available on `update_template` / `put_container` — appropriate for the record's *owning* host, which knows the whole bag.
- The targeted operations `patch_template_metadata` / `patch_container_metadata`, which address **one** namespace block: `{namespace, patch}` applies an RFC 7386 JSON Merge Patch to `metadata[namespace]` (creating the block if absent — and, in the merge-patch tradition, a `null` value removes a key), while `{namespace, de-`

`lete: true}` removes the block entirely. Exactly one of `patch / delete` is required. Nothing else in the record is touched : no sibling block, no content, no version.

The targeted operations are the safe path for shared stores where several tools cohabit on the same records. They also update the prompt's `modified` timestamp and emit a `template_updated` event, like any mutation — but they never create a version, since content is untouched.

At export time (chapter 10), `x_*` blocks travel verbatim as per-record keys of the package format, and bag keys that were never namespaced are grouped under `x_unnamespaced` so nothing is silently dropped.

Partie 2

The Template API

Template Lifecycle and Versioning

Chapter IV — Template Lifecycle and Versioning

This chapter documents every mutation and read of a prompt record: creation with its identifier derivation, the update surface with its partial-patch semantics, the metadata block patch, the trash lifecycle (delete / restore / purge), key resolution, and the versioning rules the Platform enforces uniformly over every storage adapter.

Method names below are the dispatch names (valid across C ABI, SDKs and HTTP alike); Rust signatures are in appendix A.

4.1 — create_template

Arguments — the `CreateTemplate` object: `name`, `content`, and optionally `template_id`, `description`, `collection`, `inherits`, `virtual`, `disabled`, `aliases`, `diamond_policy`, `metadata`.

Returns — a `TemplateRecord`: the full stored prompt (flattened) plus a `version` field holding the current semantic version.

The rules, in the order they apply:

1. **Identity is required.** At least one of `name` or `template_id` must be non-blank; otherwise `invalid_input`.
2. **Logical id.** If `template_id` is provided, it is canonicalized (trimmed, UPPERCASED). If not, it is *derived from the name*: whitespace and hyphens become underscores (collapsed, never leading/trailing), alphanumerics and underscores pass through, every other character is dropped, then the result is uppercased. "Code review" → `CODE_REVIEW`; "état - 2!" → `ÉTAT_2`. A derivation that ends up empty is `invalid_input`.
3. **Uniqueness.** A live prompt with the same `template_id` (case-insensitive) in the same `collection` is a `conflict`. Trashed templates do not block reuse of their id.
4. **Server-assigned fields.** The Platform mints the `uid` (UUID v4), stamps `created` and `modified` (RFC3339 UTC), and forces `deleted: false`, `deleted_at: null`, `deprecated: null`. An empty-string `collection` or `inherits` is normalized to `null`.
5. **First version.** The content is snapshotted as version `0.1` with operation `create`.
6. **Event.** One `template_created` event is emitted after commit.

```
rec = platform.create_template({
  "name": "Welcome",
  "content": "@@name Welcome\n\nWelcome aboard, @@=USER.",
  "collection": "Marketing",
```

```

}))
rec["uid"]          # "3f2a..."      - storage identity
rec["template_id"]  # "WELCOME"       - logical identity
rec["version"]      # "0.1"

```

4.2 — update_template

Arguments — the target `uid` plus an `UpdateTemplate` patch. Every field is optional; only provided fields change. Two riders control versioning (`major_version`, `version_note`) and one marks the save as a rollback (`restore_operation`).

Returns — the updated `TemplateRecord`.

Partial-patch semantics, field by field:

- `name` — applied only if non-blank (a blank name is ignored, not an error).
- `content` — replaces the source. Whether it *changed* (compared to the stored content) decides whether a version is appended.
- `template_id` — canonicalized; applied only if the canonical form is non-empty and different.
- `description`, `virtual`, `disabled`, `aliases`, `diamond_policy`, `metadata` — replaced as provided. Note that `metadata` here is the **wholesale** bag replace, reserved by convention to the record's owning host (chapter 3); for shared stores use `patch_template_metadata`.
- `collection`, `inherits`, `deprecated` — *double-option* fields: in JSON, omitting the key leaves the value unchanged, while an explicit `null` clears it (moves to root / removes inheritance / clears deprecation). An empty string behaves like `null`.

Ordering guarantees worth relying on:

- **Validation precedes mutation.** A major bump without a note is rejected *before* anything is written — a failed update leaves the store untouched.
- **The id-conflict check runs against the patched record** (excluding itself), so re-names and moves are checked in their destination collection.
- `modified` is restamped on every successful update; `created` never changes.
- One `template_updated` event is emitted, whether or not content changed. A version-append additionally emits `version_appended`.

4.3 — The versioning rules

All version bookkeeping lives in the Platform core — adapters only persist what they are told. The complete rule set:

- **First entry** is always `0.1` (operation `create`).
- **Identical content appends nothing.** Identity is checksum-based: SHA-256 over the content with `\r\n` and `\r` normalized to `\n`, truncated to the first 8 hex characters — so a save that only changes line endings is « identical ».
- **Minor bump** (default): `x.y` → `x.(y+1)`.

- **Major bump** (`major_version: true`): `x.y` $\rightarrow$ `(x+1).0`, and `version_note` is **mandatory** — `invalid_input` otherwise.
- **Operations**: `create`, `update`, `restore` (set by passing `restore_operation: true` on the update that re-saves an old content), `migrated` (reserved for migration tooling).
- Each entry records `version`, `operation`, `datetime` (RFC3339 UTC), the optional `note`, and the `checksum`; the full content snapshot is stored alongside, retrievable exactly.

The read side :

- `list_versions {uid}` — the entries, oldest first. A prompt whose history was purged returns an empty list.
- `get_version_content {uid, version}` — the exact snapshot for "`0.1`"-style version strings; `not_found` if the version does not exist.

```
platform.update_template(uid, {"content": v2})           # → 0.2
platform.update_template(uid, {"content": v2})           # no-op: still 0
platform.update_template(uid, {"content": v3, "major_version": True,
                               "version_note": "big rewrite"}) # → 1.0
old = platform.get_version_content(uid, "0.1")
# Roll back:
platform.update_template(uid, {"content": old, "restore_operation": True}) # → 1.1 (
```

4.4 — patch_template_metadata

Arguments — `uid`, `namespace` (a block key such as "`x_myapp`"; any non-empty string is accepted), and exactly one of `patch` (an RFC 7386 JSON Merge Patch applied to `metadata[namespace]`, block created if absent) or `delete: true` (remove the block).

Returns — the updated `TemplateRecord`.

The operation touches nothing but the addressed block: sibling `x_*` blocks, content and versions are untouched; `modified` is restamped and one `template_updated` event fires. A bag that is currently `null` is promoted to `{}` first; a bag that is a non-object JSON value is refused (`invalid_input`). Merge-patch semantics are the standard ones: objects merge recursively, non-objects replace, and a `null` value deletes its key.

```
platform.patch_template_metadata(uid, "x_myapp", {"pinned": True, "old_flag": None})
platform.patch_template_metadata(uid, "x_myapp", delete=True) # explicit user action
```

4.5 — The trash lifecycle

Deletion is soft by default and reversible until purged :

- **delete_template {uid}** — sets `deleted: true` and stamps `deleted_at`. The prompt disappears from key resolution and default listings, but keeps every field —

including its `collection`, so a restore puts it back where it was. Emits `template_deleted`.

- **restore_template {uid}** — clears the flags, restamps `modified`, emits `template_restored`.
- **purge_template {uid}** — hard removal: the record is deleted from storage **and its entire version history is purged**. `not_found` if the uid does not exist. Emits `template_purged`. There is no undo.

Two behaviors to note. First, trashed templates remain reachable by uid through the repository (and are visible to `list_templates` with `include_deleted: true` and to `scan` — chapter 9), which is how a host builds a trash UI. Second — a deliberate registry semantic explained in chapter 8 — trashed and disabled templates **are still part of the engine registry**: top-level checks are an application concern, and the historical contract keeps the engine's world stable regardless of application-level state.

4.6 — get_template and key resolution

get_template {key} resolves liberally and returns a `TemplateRecord` :

1. exact `uid` match, if that prompt is live ;
2. otherwise a scan of live templates for a case-insensitive `template_id` match, then an alias match — first hit in enumeration order wins ;
3. otherwise `not_found`.

The same resolution backs every key-taking read (`render_template`, `validate_template`, `analyze_template`, `list_dependencies`, `include_candidates`). Because a bare `template_id` may exist in several collections, hosts working across collections should either address by uid or keep ids unique themselves — resolution does not guess, it takes the first match in the adapter's stable enumeration order (`created` then `uid` in the shipped adapters).

`resolve_collection {path}` (chapter 5) and the enumeration primitives (chapter 9) cover bulk reads.

4.7 — TemplateRecord

Every CRUD call that returns a prompt returns it in the same envelope: the `StoredTemplate` fields flattened at the top level, plus `version` — the latest entry of the version history, or `null` when none exists (possible after a purge). The JSON shape is spelled out in appendix B.

Containers and Organization

Chapter V — Containers and Organization

Containers give a tenant's prompt store its shape: folders and projects for organization, collections for namespacing and inheritance, libraries for packaging. This chapter covers the container API — upsert with its validation rules, the metadata block patch, deletion with its cascade guard — and the exact semantics of each kind. How containers translate into the engine's world (qualification, transparency, scope markers) is chapter 8.

5.1 — The container row

A container is a metadata row keyed by slash path, per tenant:

Field	Type	Notes
kind	folder collection library project	
description	string	
label	string or null	Display label (distinct from the path leaf).
inherits	string or null	OO ² parent path — collections only.
requires	array of strings	Export dependencies — libraries only; uppercased, deduplicated.
metadata	JSON value	Opaque host extension bag.

Paths are plain slash paths ("Team/Reviews"). Rows are independent of each other and of templates: a prompt may reference a path that has no row (the path is then transparent — an undeclared folder), and a row may exist with no member templates.

5.2 — list_containers and resolve_collection

- **list_containers** — every container row of the tenant, as an object keyed by path, in deterministic (lexicographic) order.
- **resolve_collection {path}** — the live templates whose `collection` is exactly `path` or lives under it (`path/...`), as summaries. This is a convenience over the enumeration primitive `list_templates {collection_prefix: path}` with an effectively unbounded page.

5.3 — put_container

Arguments — `path` plus a `ContainerPatch`: `kind`, `description`, `label`, `inherits`, `requires`, `metadata` — all optional. **Returns** — the resulting `ContainerMeta`.

`put_container` is an upsert with patch semantics: an existing row is loaded and only provided fields change; a new row starts from defaults. Two kind-defaulting rules matter: a **new** row with no explicit `kind` is created as a `collection` (the common case), and an existing row's `kind` is only changed when `kind` is provided. `label` and `inherits` are double-option fields — JSON `null` clears them, omission leaves them alone; empty strings behave like `null`.

Validation, in order, all before anything is written:

1. **Path validity** — non-empty, and every segment non-blank, not `.`, not `..`.
2. **Nesting** — checked against the *declared* ancestors of the path (undeclared segments behave as folders):
3. a `collection` may nest under anything;
4. a `folder` or `project` may **not** live inside a `collection` or a `library`;
5. a `library` may **not** live inside a `collection` or another `library`. Violations are `invalid_input` with reason `INVALID_NESTING: ...`.
6. **Collection inheritance** (`inherits` provided and non-null):
7. only `collections` may inherit — `INHERITS_ON_NON_COLLECTION` otherwise, and the *target* must exist and be a `collection` too (`not_found` / `INHERITS_ON_NON_COLLECTION`);
8. self-inheritance and any cycle through the staged `inherits` graph are `cyclic_reference` errors, with the offending path chain in the error.
9. **Library rules** (`kind` is `library`):
10. the path **leaf** must be a valid library name — `[A-Za-z][A-Za-z0-9_]*` — else `INVALID_LIBRARY_NAME`; the canonical name is its UPPERCASE form;
11. `requires` entries are trimmed, uppercased and deduplicated; each target must exist as a `library` (`not_found`); self-reference and cycles through the staged `requires` closure are refused (`CIRCULAR_LIBRARY_DEPENDENCY`);
12. conversely, a non-empty `requires` on a non-library is `invalid_input` (and providing `requires` on a non-library clears it).

On success one `container_changed` event fires, which also invalidates the tenant's registry cache — a kind change (say, `folder` → `collection`) immediately changes how member ids are addressed at render time.

```
platform.put_container("Marketing", {"kind": "collection", "description": "Customer-fac
platform.put_container("Marketing/Onboarding", {"kind": "collection", "inherits": "Mark
platform.put_container("STDLIB", {"kind": "library", "requires": ["CORE"]})
```

5.4 — patch_container_metadata

The container twin of the prompt operation (chapter 4): `{path, namespace, patch | delete}` applies an RFC 7386 merge-patch to — or removes — exactly one `x_*` block of the container’s bag, leaving siblings and every other field untouched. The path must exist (`not_found` otherwise; unlike `put_container` this never creates a row). One `container_changed` event fires.

5.5 — delete_container

Arguments — `path` and a `cascade` mode: `block` (default) or `detach`.

Returns — a `ContainerDeletion` report: `trashed_uids` (templates soft-deleted), `removed_paths` (container rows removed), `detached_inherits` (collections whose inheritance was cut).

Deleting a container is a subtree operation with an OO^2 guard :

1. **The guard.** If any collection *outside* the subtree inherits from the path or from one of its sub-paths, the deletion is refused in `block` mode (`conflict` with reason `INHERITS_DESCENDANTS_BLOCK_DELETE: <paths>`), because it would orphan the inheritance axis. In `detach` mode those collections have their `inherits` cleared instead, and are reported in `detached_inherits`.
2. **Members are trashed, not purged.** Every live prompt whose `collection` is the path or under it is soft-deleted — it keeps its `collection` field, so restoring the prompt later puts it back in place. The uids are reported in `trashed_uids`.
3. **Rows are removed** for the path and every sub-path that had one.

One `container_deleted` event fires for the operation as a whole (the individual prompt soft-deletes do not emit separate `template_deleted` events — the container event is the signal).

Note the asymmetry with prompt deletion : container rows are hard-removed (there is no container trash), while member templates go to the ordinary prompt trash and can be restored individually. Restoring the *structure* means re-declaring the container rows; restored templates then land back under them.

5.6 — The kinds in practice

Folder — pure organization. Invisible to addressing, no engine artifact. Use folders (declared, or simply undeclared path segments) to keep large stores tidy without affecting identifiers.

Project — organization plus a resolution boundary. Like a folder for addressing, but the engine is told where projects start (scope-boundary markers, chapter 8), which bounds include resolution: templates inside a project do not silently resolve includes against unrelated templates outside it.

Collection — the namespacing and OO² unit. Member ids become qualified (`Marketing.WELCOME`); nested collections produce dotted names (`Top.Inner`); a collection may inherit from another collection, giving its members access to the parent's members along the language's rules (member shadowing, `parent` , virtual members — engine semantics, summarized in chapter 8).

Library — the packaging unit. A strict single-identifier name (canonically UPPERCASE), an explicit dependency list (`requires`), and — in the engine registry as at export — its child collections and members are declared as one namespace. Libraries exist to travel: chapter 10's package format exports them as first-class objects.

Rendering

Chapter VI — Rendering

Rendering is the Platform’s reason to exist: turn a stored prompt plus host-supplied inputs into final text, through the one reference engine. This chapter documents the render request in full — parameters, system variables, section toggles, constants, attachments, parameter sets, frames — the three render entry points, the outcome shape, and the resilience behavior when a tenant’s store contains broken sources.

6.1 — The pipeline in one paragraph

`render_template(tenant, key, request)` resolves the key (chapter 4), refuses disabled templates (`conflict`), then obtains the tenant’s *world*: the engine registry built from every stored prompt and container row (cached per tenant — chapter 8). The stored source is qualified — its `@@@template` header aligned with the record’s identity and inheritance — the request is translated into an engine `Context`, and `pool_core::render(source, context, registry)` produces either a rendered result mapped to a `RenderOutcome`, or a blocking diagnostic mapped verbatim to an `engine_diagnostic` error. The Platform adds no grammar semantics of its own on this path; what it adds is the world construction, the context routing, and one safety net described at the end of this chapter.

6.2 — The `RenderRequest`

All fields are optional; maps are string-to-string unless noted:

Field	Purpose
<code>parameters</code>	User parameter values, keyed by declared parameter name.
<code>system</code>	Host-computed system-variable values (<code>DATE</code> , <code>USER</code> , ...). The Platform never computes these itself.
<code>section_toggles</code>	Map of section name → bool, driving togglable sections.
<code>constants</code>	Fixed-value inputs that behave as defaults: explicit parameters win.
<code>attachments</code>	Runtime content for <code>@@@attachment ... : ask</code> bindings, keyed by attachment name.
<code>parameter_set</code>	Named parameter set to activate.
<code>include_depth</code>	Override of the engine’s include-depth limit.

Field	Purpose
frames	Per-included-frame inputs, keyed by dotted frame path — each a {parameters, section_toggles, parameter_set} object.

System variables and routing precedence

Twelve reserved names are routed to the engine's *system* context rather than to parameters:

DATE, TIME, DATETIME, YEAR, MONTH, DAY, WEEKDAY, USER, USERNAME, LANG, GROUPS, GROUPS_ADMIN

The design principle: **the host computes the values, the Platform routes them.** Your backend decides what DATE means (which timezone, which clock) and what USER is (session identity); the Platform passes the values through so `@@@=DATE` substitutes them. (USERNAME is an application-level alias of historical origin; the engine tolerates it as an extra system entry.)

The context is assembled with a precise precedence, worth knowing because it is a behavioral contract:

1. `system` entries seed the system map.
2. `parameters` are split by name: a key matching a reserved system name goes to the **system** map — *overriding* the host-supplied value (historical contract: an explicit parameter wins) — anything else goes to parameters.
3. `constants` fill only *gaps*: a constant whose key is already present (as parameter or as system value) loses. Constants behave as fixed-value defaults, not overrides.
4. `section_toggles`, `attachments`, `parameter_set` and `include_depth` map across directly; `frames` populate the engine's per-frame inputs.

Frames

When a prompt includes other templates, each include creates a *frame*, addressed by a dotted path. The `frames` field lets the host target inputs at a specific included frame — parameters, toggles or a parameter set that apply inside that include only, distinct from the root inputs. Frame paths and their semantics are engine territory (§6.1 of the language reference); the Platform transports them untouched.

6.3 — The RenderOutcome

```
{
  "text": "...the rendered prompt...",
  "warnings": [ { "code": "...", "message": "...", "handle": null } ],
  "attachments": {
    "included": [ { "name": "spec", "type": "pdf", "kind": "included", "frame": "" } ],
    "to_include": [ { "name": "logs", "type": "text", "kind": "to_include", "frame": "r" } ]
  }
}
```

```
}  
}
```

- **text** — the engine’s rendered output, byte-exact (the conformance parity gate compares it byte-for-byte against the standard’s expected outputs). It carries the prompt’s declared structure as markdown headings — `##` per section, `### <n>`. per sub-section (§6.1 PHASE 11 of the language reference); the prompt itself contributes none, its name and id travelling on the value rather than in the text. The Platform transports it untouched; it never adds or strips a heading of its own.
- **warnings** — the engine’s non-blocking diagnostics, transported as data: `code` is the engine code verbatim, `message` the engine’s full string. The Platform deliberately does **not** replicate any application’s warning-promotion policy (deciding that some warnings should fail a save is a host decision, applied over this list). One non-engine code can appear here — `INVALID_REGISTRY_ENTRY`, described below.
- **attachments** — the attachment declarations the render traversed, aggregated across includes and split by kind: `included` (stored content, bytes retrievable via the blob store) versus `to_include` (runtime requirement descriptors). Each carries the frame path it was declared in (`"` = root).

A *blocking* diagnostic does not produce an outcome: it becomes the typed error `engine_diagnostic` with the engine’s `SCREAMING_SNAKE code`, its `message`, a best-effort `handle` (the offending identifier extracted from the message), and the pipeline phase when the spec pins one. The Platform keeps the engine’s choice of *earliest blocking diagnostic* — the conformance identity.

```
try:  
    out = platform.render_template("REVIEW", {  
        "parameters": {"language": "rust"},  
        "system": {"USER": "alice", "DATE": "2026-07-12"},  
        "section_toggles": {"examples": False},  
    })  
except PlatformError as e:  
    e.kind    # "engine_diagnostic"  
    e.code    # e.g. "UNDEFINED_REFERENCE"
```

6.4 — render_source — previews and unsaved text

`render_source {source, request, as_template?}` renders raw text against the tenant’s registry without storing anything — the editor-preview primitive.

Without `as_template`, the source renders as an anonymous root document: it can include anything the tenant’s world exposes at root scope. With `as_template: <key>`, the source is *qualified as if it were the stored content of that prompt*: it inherits the prompt’s collection qualification and `inherits` line, and its caller folder is the prompt’s folder path — so `self.... / parent....` references and folder-relative includes resolve exactly as they will after saving. This is the difference between «render this text» and «preview this edit of prompt X», and the second is what an editor wants.

The registry the source renders against is the *stored* world: the prompt's saved content is still in the registry (the preview source is rendered as an additional document, it does not replace the stored entry for includes that other paths might take).

6.5 — render_with_registry — explicit worlds

The third entry point, `render_with_registry(source, request, entries, caller_folder)` (Rust-level; not exposed through the JSON dispatch), renders against an explicitly supplied registry with no store involved. Paired with `build_registry` (chapter 8) it supports advanced hosts that snapshot a world and replay renders against it — deterministic test harnesses, offline batch rendering, or caching layers of their own.

6.6 — Disabled templates

A disabled prompt refuses to render: `render_template` returns a `conflict` («prompt is disabled») *before* touching the engine. Disabling is a top-level check only — a disabled prompt remains in the registry and therefore still resolves when *included by others* (chapter 8 explains why the registry ignores application state). A host that wants disabled-means-uninlinable enforces that policy itself, e.g. by checking `list_dependencies` before rendering callers.

6.7 — The poisoned-registry fallback

One broken source in a tenant's store must not take every render down. The engine tolerates invalid registry entries that a render never references; the failure mode needing a net is a *parse-poisoned* world where an entry the pipeline does touch fails at phase 1.

The Platform's fallback: when a render fails with a phase-1 (parsing) diagnostic **and** the entry source itself is valid, every registry entry is re-validated individually; if culprits exist, they are excluded and the render retried once. A successful retry returns normally, with one synthesized warning per excluded entry:

```
{ "code": "INVALID_REGISTRY_ENTRY",  
  "message": "INVALID_REGISTRY_ENTRY: prompt 'BAD' failed to parse (...); excluded from t  
  "handle": "BAD" }
```

`INVALID_REGISTRY_ENTRY` is the only warning code the Platform synthesizes (it is not an engine code). Project scope markers are always kept during the sweep. If the retry still fails — or the original diagnostic was not parse-phase — the engine's diagnostic is returned as usual. The net effect: a tenant can carry a half-written prompt without bricking unrelated renders, and hosts see exactly which entry was excluded.

6.8 — Differences from the historical adapter

For readers coming from the IziBuddy codebase, two deliberate differences from the Python `pool_adapter`, both documented decisions: there is no «stale engine binding» fallback and no empty-render guard (the Platform links the engine in-process — a stale binary cannot exist), and engine diagnostic codes are transported **verbatim** in SCREAMING_SNAKE (the historical lowercasing was an application concern and stays one).

Validation, Analysis and Dependencies

Chapter VII — Validation, Analysis and Dependencies

Beyond rendering, the Platform exposes the engine's static services — parse-validation and structural analysis — plus two store-aware derivatives: dependency listing with resolution flags, and include-candidate discovery for editor autocompletion. All of them are pure passthroughs or thin compositions over the engine; the Platform reinterprets nothing.

Each service exists in two forms: a `*_template {key}` variant that resolves a stored prompt and operates on its content, and a `*_source {source}` variant that operates on raw text (unsaved editor state). Both use the same underlying engine call.

7.1 — `validate_template` / `validate_source`

Validation is parse-only: no rendering, no reference resolution against the store. It never *fails* — grammar errors land in the result:

```
{ "ok": true, "kind": 'template', "warnings": [ "..." ], "errors": [] }
{ "ok": false, "kind": null, "warnings": [],
  "errors": [ { "code": "DUPLICATE_DECLARATION", "message": "...", "phase": 1 } ] }
```

- **ok** — whether the source parses.
- **kind** — what the source is: `prompt`, `collection`, `library` or `unstructured`; absent on a blocking parse error.
- **warnings** — non-blocking messages (strings).
- **errors** — blocking diagnostics as `{code, message, phase?}` objects, engine codes verbatim.

(One asymmetry: `validate_template` can still return `not_found` — resolving the key can fail even though validation itself cannot.)

7.2 — `analyze_template` / `analyze_source`

Analysis returns the static structure of a source — everything a frontend needs to build an instantiation form, a declaration editor or a structure view, without a hand-rolled grammar parser. A blocking grammar error is returned as the typed `engine_diagnostic` error; success returns:

Field	Content
<code>kind</code>	<code>prompt</code> <code>collection</code> <code>library</code> <code>unstructured</code> .
<code>id</code> , <code>name</code> , <code>description</code> , <code>inherits</code>	Header identity, when declared in the source (<code>id</code> may be dotted-qualified).
<code>virtual</code> , <code>final</code>	Header modifiers.
<code>params</code>	Declared parameters — see below.
<code>consts</code>	Declared constants: <code>{name, type, value}</code> each.
<code>sections</code>	Sections: <code>{name, description?, togglable, default_on, virtual, final, delete, transparent, placement?}</code> each. <code>delete</code> is <code>@@@section delete <name></code> (renders nothing, heading included, yet stays addressable); <code>transparent</code> is <code>@@@section (<name>)</code> (no heading of its own, sub-sections emit their rank alone); <code>placement</code> is the declared clause, <code>{kind, anchor?}</code> with <code>kind</code> one of <code>first</code> / <code>last</code> / <code>before</code> / <code>after</code> .
<code>includes</code>	Include references: <code>{handle, args, transparent, has_args, section_after, candidates}</code> each — <code>handle</code> is the referenced identifier, <code>args</code> the literal call arguments as name/value pairs, <code>transparent</code> says the reference was wrapped (<code>@@@include (<reference>)</code>), the consumer-side half of <code>@@@section (<name>)</code>), <code>section_after</code> the section path selected after the call if any, and <code>candidates</code> every target the include can reach READ FROM THE SOURCE ALONE — one entry for a written target, the declared domain of the symbol for an indirect one. Do not confuse <code>candidates</code> with the <code>include_candidates</code> method, which answers a different question: what this prompt could legally include, resolved against the tenant registry.
<code>warnings</code>	Non-blocking messages (strings).

Each entry of `params` describes one declared parameter:

```
{ "name": "note", "type": "text", "default": null,
  "select_options": [], "min": null, "max": null,
  "minlen": null, "maxlen": null, "pattern": null,
  "optional": true, "required": false, "label": "Note" }
```

- **type** — the declared type; for `select` / `multiselect` parameters, `select_options` lists the choices.
- **default** — the declared default value, if any.

- **constraints** — `min / max` (numeric), `minlen / maxlen / pattern` (text), flattened into the object; absent when undeclared.
- **optional** — `true` when the parameter is declared with the optional marker (the `?` suffix on the keyword: `@@@param? note : text`). Per the language (§5.6) optionality is *independent of the default*: a non-optional parameter may still carry a default. **required** is the derived convenience `!optional`.
- **label** — the human-readable label from a trailing `, "..."` in the declaration, including the no-default form `@@@param x : type, "label"`.

A worked example :

```
a = platform.analyze_source(
  '@@@template review\n'
  '@@@param lang : text = "fr", "Language"\n'
  '@@@param? note : text, "Note"\n'
  '@@@const MAX : number = 100\n'
  '@@@section main\n'
  '@@@=lang (max @@@=MAX)\n'
)
a["params"][0] # {"name": "lang", "type": "text", "default": "fr",
                # "optional": false, "required": true, "label": "Language", ...}
a["params"][1] # {"name": "note", ..., "optional": true, "required": false, "label": "..."}
a["consts"]    # [{"name": "MAX", "type": "number", "value": "100"}]
```

7.3 — list_dependencies

`list_dependencies {key}` answers « what does this prompt point at, and does it resolve here? » — the store-aware layer over analysis. The result is a flat list :

```
[ { "handle": "PARENT",          "relation": "inherits", "resolved": true },
  { "handle": "Common.DISCLAIMER", "relation": "include",  "resolved": true },
  { "handle": "MISSING",          "relation": "include",  "resolved": false } ]
```

The `inherits` entry (at most one) comes from the stored record's field; the `include` entries come from analyzing the content. The `resolved` flag is a *presence check* against the tenant's known identifiers — every prompt's terminal id, uid and aliases, matched case-insensitively, with a dotted handle also matched by its last segment. It deliberately does **not** replicate the engine's full scoping rules: `resolved: false` reliably means « nothing by that name exists in this tenant », while `resolved: true` means « a target of that name exists » — whether *this* prompt can legally reach it (project boundaries, collection scoping) is decided by the engine at render time. Use it for dangling-reference warnings, not as a render predictor.

7.4 — include_candidates

`include_candidates {key}` returns what the prompt could legally `@@@include` from where it stands — the autocomplete primitive :

```
[ { "handle": "Common.DISCLAIMER", "aliases": ["DISCLAIMER"],
  "kind": 'template', "target": "Common.DISCLAIMER" },
  { "handle": "Common", "aliases": [], "kind": "collection", "target": "Common" } ]
```

One entry per WORKING SPELLING: `handle` is what the user types after `@@@include`, `target` the prompt it resolves to, and `aliases` the other spellings reaching that same target (the qualified path and, from inside the collection, the bare name).

Unlike `list_dependencies`, this one is scope-exact: the Platform calls the engine's `scope_candidates` with the prompt's qualified source, the tenant's registry, the prompt's caller folder and its qualified id — so project boundaries, collection qualification and folder relativity are all honored by construction. **The engine arbitrates every spelling itself**, so the prompt is excluded from its own list there and the host has nothing left to filter.

7.5 — Direct engine access

Integrators who want the raw language contract — for tooling beyond the Platform's services — get the entire `pool-core` public API re-exported verbatim as `pool_platform_core::engine` (Rust embedding only). The health endpoint of the HTTP server and the Rust API also expose the engine's implemented conformance level (`engine_implemented_level()`), useful for feature-gating hosts against future language levels.

Partie 3

Data and Registry Services

The Registry and the Cache

Chapter VIII — The Registry and the Cache

The engine knows nothing about storage: it renders a source against a *registry* — an ordered list of (key, source) entries describing the whole world of templates, collections and libraries. The Platform's central piece of machinery is the construction of that registry from a tenant's stored templates and container rows. This chapter documents the construction rules precisely (they are a semantic contract, ported verbatim from the historical shell and locked by the 511-case conformance parity), the per-tenant cache that makes rendering fast, and the inspection and invalidation surface.

Most hosts never need this chapter to *use* the Platform — rendering just works. Read it when you need to reason about identifier resolution, when you implement host storage, or when you snapshot registries for advanced workflows.

8.1 — What the builder starts from

Registry construction takes two inputs, both read through the storage SPI: every stored prompt of the tenant (live *and* trashed, enabled *and* disabled — see «Application state» below), in the adapter's stable enumeration order; and every container row. From the container rows plus the paths implied by templates' `collection` fields, the builder classifies **every path involved** — including undeclared ancestors and inheritance targets, discovered transitively.

The classification decides one thing per path segment: does it *qualify* identifiers? A segment qualifies only if its path has a **declared** row of kind `collection` or `library`. Declared folders and projects, and every undeclared segment, are transparent.

From this classification derive the two projections used everywhere:

- the **engine collection name** of a path — the dotted join of its qualifying segments: with `Team` a folder and `Team/Coll` a collection, path `Team/Coll` has engine name `Coll`; with both `Top` and `Top/Inner` collections, `Top/Inner` is `Top.Inner`; a fully transparent path has none;
- the **folder path** — the slash join of the complement (the transparent segments): `Team` in the first example. It becomes the engine's *caller folder* for templates stored there, which is what makes folder-relative sibling includes work.

8.2 — Qualifying a prompt’s source

Each stored prompt contributes one registry entry. Its **key** is `folder_path/qualified_id` (or just the qualified id at root), where the qualified id is `EngineCollectionName.TERMINAL` — the terminal id being `template_id`, falling back to the uid for the rare id-less record. Its **source** is the stored content, *qualified*:

- The `@@@template` header is aligned with the record: id replaced by the qualified id, `inherits` replaced by the parent’s qualified id — the model’s fields win over whatever the content declared. Only a leading `virtual / final` modifier of an existing header survives the rewrite.
- The parent’s qualified id is found by scanning for the **first** prompt whose terminal id equals the record’s `inherits` (enumeration order), and qualifying it with *that* prompt’s collection. An unknown parent passes through unqualified — the engine will raise `UNDEFINED_REFERENCE` at render, which is the desired signal.
- **Root no-op**: a root prompt with no parent and at least one POO²L line directive in its content passes through byte-for-byte — the registry does not touch what does not need touching. (This byte-for-byte path is what makes conformance parity exact.)
- **Bare content**: content with no directive at all is not an includable entity for the engine, so the builder injects a header and wraps the body in `@@@section main`. This is a documented, host-friendly deviation from the historical shell (which always injected `@@@name` at save time and therefore never stored fully bare content — the Platform stores content verbatim instead). Leading standalone `@@@virtual / @@@final` lines on bare content are consumed into the header (last one wins), falling back to the record’s `virtual` flag; a virtual prompt whose body has no section keeps only the header line.

8.3 — Synthesized container documents

Declared collections and libraries also contribute entries of their own — the declarative documents the engine’s containment and inheritance model consumes:

- `__collection__<Name>` — for each declared collection with an engine name:

```
@@@collection Top.Inner [inherits Parent] @@@template MEMBER_A ← member
terminal ids, enumeration order @@@collection Child ← direct public sub-
collections, sorted @@@end-collection
```

The `inherits` clause is the *engine name* of the collection’s `inherits` path.

- `__library__<NAME>` — same idea for libraries, with two differences: `@@@requires <LIB>` lines come first, and child collections are listed **before** members.
- `__project__<folder>` — for each declared project, a scope-boundary marker: an entry with that key and an *empty source*. This is how the engine learns where project boundaries lie without projects qualifying names.

Duplicate keys across the whole build **overwrite silently — last wins** (this mirrors the historical contract; with per-collection id uniqueness enforced at write time, genuine duplicates only arise from same-id templates at the same transparent path).

8.4 — Application state is not the registry’s business

Deleted and disabled templates **are in the registry**. The contract, verbatim from the historical shell: top-level checks are an application concern. `render_template` refuses disabled templates and does not resolve trashed ones as entry points — but as *include targets* both still exist, so trashing a shared block does not instantly break every caller in unpredictable ways. A host that wants stricter semantics builds them above the Platform (chapter 7’s `list_dependencies` is the tool). Case handling follows the same philosophy: the builder does no case normalization — canonical UPPERCASE ids come from the write path, and the engine canonicalizes handles itself.

8.5 — The per-tenant cache

Building the world costs a full scan, so the Platform caches it per tenant: the registry entries, the prompt list, the container map, and a per-uid index of (qualified source, caller folder, qualified id) used by `render_template` and `include_candidates`.

The invalidation rule is simple and total: **every mutation invalidates the tenant’s cache**. Each successful mutation increments the tenant’s sequence number (the same one events carry — chapter 9) and drops the cached world; the next render rebuilds it. Reads never invalidate. The cache is built outside the tenant lock (storage may be slow) and double-checked on insertion, so concurrent renders during a rebuild are safe.

Two consequences for hosts:

- **Writes through the Platform are always coherent** — there is no stale-render window after a mutation returns.
- **Writes behind the Platform’s back are not seen.** A host that mutates the underlying store directly (possible in cohabitation setups, or with host storage) must call `invalidate` (`{tenant} → {done: true}`) to force a rebuild. This is the one cache-management method in the API.

8.6 — `build_registry` — inspecting the world

`build_registry` returns the tenant’s registry as data:

```
{ "entries": [ ["__collection__Common", "@@@collection Common\n..."],  
               ["Common.DISCLAIMER", "@@@template Common.DISCLAIMER\n..."],  
               ["Team/REVIEW", "@@@template REVIEW inherits Common.BASE\n..."] ],  
  "seq": 42 }
```

`entries` is the exact ordered list the engine would receive; `seq` is the tenant sequence the snapshot was built at — compare it with event sequences to know whether a snapshot is current. Uses: debugging resolution surprises («what does the engine actually see?»), host-side caching keyed by `seq`, and — in Rust — replaying renders against a frozen world via `render_with_registry` (chapter 6).

Enumeration, Events and Blobs

Chapter IX — Enumeration, Events and Blobs

The Platform ships *data, never opinions*: instead of a search engine, an export format zoo or an indexing service, it provides three primitives from which a host builds all of those its own way — field-selected, cursor-paged enumeration; a lossless full-fidelity scan; and exactly-one-per-mutation business events with durable per-tenant sequences. Blobs complete the picture as the storage side of attachments.

9.1 — list_templates — the enumeration primitive

Arguments — a `ListQuery` ; every field optional :

Field	Effect
<code>collection_prefix</code>	Only templates whose <code>collection</code> equals the prefix or lives under it (<code>prefix/...</code>). Omitted = everywhere ; empty string = root only (templates with no collection).
<code>include_deleted</code>	Include trashed templates (default false).
<code>uids</code>	Restrict to these uids (exact match).
<code>modified_since</code>	RFC3339 lower bound on <code>modified</code> (inclusive ; string comparison — timestamps are normalized RFC3339 UTC, so it is a correct time order).
<code>fields</code>	Field selection : <code>summary</code> (default — metadata only), <code>with_content</code> (adds <code>content</code>), <code>full</code> (adds the metadata bag too).
<code>cursor</code>	Opaque cursor from a previous page ; omitted = from the beginning.
<code>limit</code>	Page size ; 0 or absent = adapter default (500).

Returns — a page: `{items: [TemplateSummary...], next_cursor}` . `next_cursor` is present exactly when more items exist; feed it back verbatim. Treat cursors as opaque and non-portable across mutations (they index into the stable enumeration).

`TemplateSummary` carries the listing fields (`uid`, `template_id`, `name`, `description`, `collection`, `inherits`, `virtual`, `disabled`, `deleted`, `created`, `modified`) plus `content` and/or `metadata` when the query asked for them — absent keys, not nulls, otherwise.

Enumeration order is deterministic and adapter-stable — `created` then `uid` in the shipped adapters — and the filter/selection/pagination semantics are *reference semantics*: a shared implementation defines them, the shipped adapters implement them identically (verified by a common behavioral suite), and host storage gets them computed in Rust over its scan (chapter 12). Whatever the backend, `list_templates` behaves the same.

```
# Incremental sync: everything modified since the last sync point.
for s in platform.iter_templates(modified_since=last_sync, include_deleted=True, fields=fields):
    reindex(s) # iter_templates = the SDK's cursor-following loop over list_templates
```

9.2 — scan_templates — the lossless read

`scan` streams **every** prompt of a tenant — live and trashed — as complete records, in the same stable order, with early stop supported. It is the fidelity contract: `scan` + `list_containers` + `list_versions` + `get_blob` reconstruct a tenant *without loss*, including opaque metadata bags (returned verbatim, so foreign `x_*` blocks survive round-trips) and provenance timestamps. Any host-built export starts here; the Platform's own package export (chapter 10) is built on exactly this surface — the dogfooding rule in action.

Over the JSON surfaces, the same result is reached with `list_templates {fields: "full", include_deleted: true}` and cursor-following; `scan` itself is the Rust/SPI-level streaming form.

9.3 — Events — the incremental contract

Every successful mutation emits **exactly one** business event, after commit:

Event type	Payload	Emitted by
<code>template_created</code> / <code>template_updated</code> / <code>template_deleted</code> / <code>template_restored</code> / <code>template_purged</code>	<code>uid</code>	the prompt lifecycle (metadata block patches emit <code>template_updated</code>)
<code>container_changed</code> / <code>container_deleted</code>	<code>path</code>	<code>put_container</code> , <code>patch_container_metadata</code> / <code>delete_container</code>
<code>version_appended</code>	<code>uid</code> , <code>version</code>	any content change that snapshots a version
<code>blob_stored</code>	<code>blob</code> (the ref)	<code>put_blob</code>

Each event carries the tenant and a **per-tenant strictly increasing sequence number**. The delivery contract: fire-and-forget after commit — a sink must never block or fail the caller; the mutation has already happened when it fires. Note that one logical operation can emit more than one event (`create_template` → `version_appended` + `template_created`), and `delete_container` emits a single `container_deleted` for the whole subtree (the member soft-deletes are implied).

Consuming events is a Rust-embedding concern: the `PlatformConfig.events` sink receives them in-process (the C ABI and HTTP surfaces do not stream events today — a host on those surfaces polls with `modified_since`, or embeds). What every surface *does* get is the sequence side: `build_registry` exposes the tenant's current `seq`, and event-driven cache invalidation is internal and automatic.

Durable sequences

Sequences exist so a host can index incrementally: remember the last `seq` processed, resume from there. That only works across restarts if sequences do not reset — so the Platform persists the counter through an optional `SeqStore` (chapter 11): seeded on first tenant access, updated best-effort on every increment (a failed persist never fails the mutation, which has already committed). The filesystem and SQLite configs wire it automatically — a per-tenant `seq` file, a `seqs` table. Memory and hosted configs keep per-process sequences; after a restart of those, an incremental indexer should re-scan.

9.4 — Blobs

The storage side of attachments, content-addressed:

- **put_blob** — stores bytes, returns the ref `sha256:<hex>` (base64-encoded bytes over JSON transport). Idempotent by construction: same bytes, same ref. Shipped adapters refcount each put. Emits `blob_stored`.
- **get_blob {ref}** — the bytes back; `not_found` for unknown refs.
- **release** (SPI level) — refcount decrement, delete-if-unreferenced. Deliberately not on the JSON dispatch: reference lifecycle belongs to the host layer that knows who points at what.

The link to rendering: `RenderOutcome.attachments.included` names stored attachments — their content lives here, addressed by ref; `to_include` entries are runtime descriptors the host satisfies per render (chapter 6).

9.5 — Building search and export on top

The intended pattern, spelled out once: a host maintains its own index (full-text, vector, tag-based — its opinion) by seeding from a `scan` and keeping current with events (embedded) or `modified_since` polling (remote), keyed by `uid` and cut off by `seq`. Export is a `scan` plus versions and blobs, serialized however the host likes — or simply chapter 10's native package when the standard format fits. The Platform's part of the bargain is the fidelity and exactly-once contracts above; ranking, matching and envelopes stay yours.

Packages : Export and Import

Chapter X — Packages : Export and Import

The Platform speaks exactly one exchange format : the POO²L standard's package (§7.4), implemented natively as `export_package` / `import_package`. It earns its place inside the « data, never opinions » rule because it is *normative* — defined by the language standard, not by any application — and implementing it once in the Platform spares every host a hand-rolled conformance effort. It is also a dogfooding exhibit : export is built entirely on the public read surface (scan, containers, the registry builder), nothing privileged.

10.1 — The design principle : content is the single source of truth

The exported record of a structured prompt restates **no structured field** : not `template_id`, not `name`, not `collection`, not `inherits`, not `virtual`, because all of them are readable from — or projected into — the POO²L content itself. What the Platform exports as `content` is the same *qualified projection* the engine registry receives (chapter 8) : the `@@@template` header carries the qualified id and inheritance, so the text alone is self-describing. Two escape hatches keep the format lossless anyway :

- **Level-0 content** (no POO²L directive at all) cannot carry identity in text, so — per the standard's identity exception — the record adds an `identity` object : `{id, name, description}`, and the content travels verbatim.
- **Platform-owned fields with no textual form** — a `name` or `description` the content does not declare, trash and disabled state, container labels — ride the Platform's own implementation block `x_poo2l_platform`, strippable like any `x_*` block.

10.2 — `export_package`

Arguments — `{include_deleted?: bool}` (default false). **Returns** — the envelope :

```
{
  "poo2l_package": "1.0",
  "exported_at": "2026-07-12T22:01:37Z",
  "templates":    [ ...records... ],
  "collections":  [ ...records... ],
  "libraries":    [ ...records... ],
  "external_dependencies": [ { "kind": 'template', "reference": "MISSING" } ],
  "x_poo2l_platform": { "folders": { "Team": { "kind": "folder", "label": "..." } } }
}
```

Template records. Each carries `storage_id` (the uid — informational; importers never match on it), `content` as described above, `folder_path` (the transparent segments of its container path, as an array — present only when non-root), the `x_poo2l_platform` block when needed (`name / description` fallbacks, `deleted / deleted_at`, `disabled`), and then the metadata bag: every `x_*` block **verbatim** as a record key, with never-namespaced bag keys grouped under `x_unnamespaced` so nothing is silently dropped.

Collection and library records. Their `content` is the synthesized declarative document from the registry builder (`@@@collection Name [inherits Parent] ... @@@end-collection`, `@@@library NAME with @@@requires` lines) — again: structure as text, not as fields. `storage_id` is a deterministic UUID derived from tenant and path (containers are keyed by path, not uid); `folder_path`, a platform block for `label / description`, and the bag follow the same rules as templates.

Folders and projects are *not* packaged entities in the standard; their display metadata rides the envelope's `x_poo2l_platform.folders` map — strippable, like everything in that namespace.

External dependencies. Every reference that resolves to nothing *inside the exported set* is listed once: includes and prompt inherits (kind `prompt`), collection inheritance targets (kind `collection`), library requires (kind `library`). Intra-template axes (`self...`, `parent...`) are never external. On a full-tenant export this list is exactly the dangling references — a useful lint even if you never ship the package.

10.3 — import_package

Arguments — `{package: <envelope>, options?: {}}`. **Returns** — a report:

```
{ "templates_created": 12, "templates_skipped": 1,
  "containers_created": 3, "containers_skipped": 0,
  "warnings": [ "prompt `WELCOME`: already exists, skipped" ] }
```

The ground rules: the envelope's `poo2l_package` version must be `1.x` (`invalid_input` otherwise); **existing entities are never overwritten** — they are skipped and reported; imported templates receive **fresh uids and a fresh version history** (POO²L references are logical, so nothing dangles because of new uids); and content is authoritative for identity, `analyze` being the reader.

The order of operations matters and is fixed:

1. **Folders and projects** from the envelope's platform block — created unless the path already exists.
2. **Collections and libraries, pass 1** — each record's content is parsed (name, inherits, requires, member list); its tenant path is `folder_path` + the dotted name's segments; missing rows are created (requires included), existing paths skipped. Inheritance wiring is *deferred*, because a parent may appear later in the package.
3. **Pass 2** — inheritance is wired now that every target exists, resolved through the dotted-name → path map (seeded with the tenant's pre-existing containers, so a package

can inherit from collections you already have). An unresolvable parent degrades to a warning, never a failure.

4. **Templates.** Identity comes from the content's header when present — a dotted id like `Marketing.WELCOME` splits into the collection run and the terminal id, and the tenant path is `folder_path` + the run's segments. Level-0 records fall back to the `identity` object; their collection placement falls back to the container documents' member listings when exactly one collection lists the id (several → warning, left at root), else to `folder_path`. Name and description resolve content-first, then `identity`, then the platform block; `virtual` comes from analysis; `disabled` from the platform block; the bag is reassembled from the record's `x_*` keys (minus the platform block) with `x_unnamespaced` restored to the bag root. An id conflict in the destination collection means skip + warning; any other error aborts the import with the underlying typed error.

10.4 — Round-trip properties

What survives `export` → `import` intact: content (structured templates byte-meaningful through the qualified projection; level-0 verbatim), logical identity, container structure with inheritance and requires, folder organization, metadata bags block-for-block, names and descriptions wherever they came from, disabled state. What intentionally does not: uids and version histories (fresh by design), trash state on default exports (`include_deleted: false` drops trashed templates; when exported, the trash flag rides the platform block for the *importing* side to interpret — the Platform's own import creates the prompt live), and anything a consumer strips by dropping `x_*` blocks it does not understand — which is precisely the standard's interoperability model.

```
pkg = platform.export_package()           # tenant → envelope
report = other_platform.import_package(pkg) # envelope → another tenant/store
assert not report["warnings"]
```


Partie 4

Storage

The Storage SPI and Shipped Adapters

Chapter XI — The Storage SPI and Shipped Adapters

Persistence is a service-provider interface : the Platform owns every semantic (versioning rules, registry construction, conflict checks, events) and delegates only the raw reads and writes to a set of small traits. Swap the traits, keep the behavior — that is the contract, and a shared behavioral suite runs against every shipped adapter to prove it. This chapter documents the traits precisely, then the three adapters in the box.

11.1 — The configuration object

A Platform is wired from a `PlatformConfig` — seven slots, each an implementation of one trait :

Slot	Trait	Default (in_memory)
templates	TemplateRepository	memory
containers	ContainerRepository	memory
versions	VersionRepository	memory
blobs	BlobStore	memory
authorization	AuthorizationProvider	AllowAll
events	EventSink	no-op
seq	Option<SeqStore>	none (per-process sequences)

All trait methods are tenant-scoped and **synchronous** (`Send + Sync`); async hosts wrap at the binding layer. A host that configures nothing gets a working single-user platform — the permissive defaults are a design statement, not an accident.

11.2 — TemplateRepository

```
fn get(&self, t: &TenantId, uid: &str) -> Result<Option<StoredTemplate>>;
fn put(&self, t: &TenantId, prompt: &StoredTemplate) -> Result<()>;
fn delete(&self, t: &TenantId, uid: &str) -> Result<bool>;
```

```
fn list(&self, t: &TenantId, q: &ListQuery) -> Result<Page<TemplateSummary>>;
fn scan(&self, t: &TenantId, f: &mut dyn FnMut(StoredTemplate) -> bool) -> Result<()>;
```

The contracts an implementation must honor :

- `get` addresses by **uid only** (logical-id resolution lives above, in the Platform).
- `put` is a full-record upsert; the Platform always writes complete records.
- `delete` is *hard* removal, returning whether the record existed (soft delete is a Platform-level `put` of a flagged record).
- `scan` streams **every** record — live and trashed — in a **stable, deterministic order**, honoring early stop. This order is load-bearing: it feeds registry construction (chapter 8), first-match key resolution (chapter 4) and pagination cursors. The shipped convention is `created` then `uid`.
- `list` must implement the reference filter/field-selection/pagination semantics of chapter 9 — exactly. The core exports the reference implementation as a helper (`paginate(all, query)`), so a custom adapter can scan into a vector and delegate, which is precisely what the host-callback layer does.

11.3 — ContainerRepository

```
fn get(&self, t: &TenantId, path: &str) -> Result<Option<ContainerMeta>>;
fn put(&self, t: &TenantId, path: &str, meta: &ContainerMeta) -> Result<()>;
fn delete(&self, t: &TenantId, path: &str) -> Result<bool>;
fn list(&self, t: &TenantId) -> Result<BTreeMap<String, ContainerMeta>>;
```

Plain keyed rows; `list` returns them in deterministic (lexicographic) path order. All nesting/inheritance validation happens above.

11.4 — VersionRepository

```
fn append(&self, t: &TenantId, template_uid: &str, entry: &VersionEntry, content: &str) -> Result<()>;
fn list(&self, t: &TenantId, template_uid: &str) -> Result<Vec<VersionEntry>>;
fn get_content(&self, t: &TenantId, template_uid: &str, version: &str) -> Result<Option<Vec<u8>>>;
fn purge(&self, t: &TenantId, template_uid: &str) -> Result<()>;
```

Append-only per prompt; `append` persists the entry *and* the exact content snapshot it refers to; `list` returns oldest-first. Deciding *whether* to append (checksum identity, bump arithmetic, the major-note rule) is the Platform’s job — an adapter never sees a rejected version.

11.5 — BlobStore

```
fn put(&self, t: &TenantId, bytes: &[u8]) -> Result<BlobRef>; // content-addressed
fn get(&self, t: &TenantId, blob: &BlobRef) -> Result<Option<Vec<u8>>>;
fn release(&self, t: &TenantId, blob: &BlobRef) -> Result<()>; // refcount decremen
```

`put` is idempotent by content address (`sha256:<hex>` in the shipped adapters) and refcounted where supported; `release` decrements and deletes when unreferenced.

11.6 — EventSink, SeqStore, AuthorizationProvider

EventSink — `fn emit(&self, event: &PlatformEvent)`, infallible fire-and-forget after commit (chapter 9). Two implementations ship: the default `NullEventSink` and a `CollectingEventSink` for tests and demos.

SeqStore — `fn load(&self, tenant) -> Result<u64>` / `fn store(&self, tenant, seq) -> Result<()>`: optional durability of the per-tenant sequence counter. Load seeds tenant state on first access; store is called best-effort after each increment — a failure never fails the mutation.

AuthorizationProvider — `fn check(&self, tenant, actor, action, key) -> Decision` with `Action ∈ {Read, Write, Delete, Render, Admin}`: the abstract permission hook. The Platform asks; the host decides; the default `AllowAll` never says no. This is the *mechanism* slot reserved for multi-actor hosts — user identity and policy remain host concerns by design.

11.7 — The memory adapter

The zero-configuration default (`Platform::in_memory()`) and the backing of the test suite: per-tenant maps behind read-write locks, insertion order preserved for templates (satisfying the stable-scan contract), refcounted blobs. Nothing survives the process — its point is development, testing and ephemeral tenants.

11.8 — The filesystem adapter

`filesystem_config(root)` wires all five slots (SeqStore included) over one directory per tenant:

```
<root>/<tenant>/
├─ templates/<uid>.json      one JSON document per prompt
├─ containers.json          all container rows of the tenant
├─ versions/<uid>.json      { "entries": [...], "contents": {version: text} }
├─ blobs/
│   └─ <sha256-hex>         raw bytes
│       └─ refcounts.json
└─ seq                      the durable sequence counter
```

Properties worth knowing: every write is **atomic** (write to a temp file, then rename); documents are pretty-printed JSON — the store is human-inspectable and diff-friendly; scan order is `created` then `uid`, recomputed from the documents; internal mutexes serialize the multi-document files (containers, versions, refcounts) within the process. The layout is append-friendly for backup tooling, and a tenant is exactly one directory — copy it, you have the tenant.

The filesystem adapter is also the natural cohabitation surface : a host may keep *its* own files next to the Platform’s tenant directory. The one rule : writes to Platform-owned files go through the Platform (or are followed by `invalidate` — chapter 8).

11.9 — The SQLite adapter

`sqlite_config(path)` — or `sqlite_config_from(SqliteDb)` over an already-open database, including in-memory — puts everything in one file with the bundled libsqlite3 (no system dependency), WAL journal mode, and a `tenant` column everywhere :

Table	Key	Payload
templates	(tenant, uid)	created + the full record as JSON
containers	(tenant, path)	the row as JSON
versions	(tenant, uid, seq)	version , the entry as JSON, the content snapshot
seqs	tenant	the durable sequence counter
blobs	(tenant, ref)	bytes + refcount

The JSON payload is the source of truth ; `list` filters in Rust over the deterministic enumeration (`ORDER BY created, uid`), which keeps semantics provably identical to the other adapters. Indexed SQL filtering is a documented optimization a dedicated adapter can add *without contract change* — the roadmap holds that slot. A single connection behind a mutex serializes access within the process (SQLite serializes writes anyway).

11.10 — The shared behavioral suite

One test suite runs the same scenario — CRUD with version bumps, containers with collection-qualified rendering, listing filters, container deletion with undo info, blobs, purge — against every shipped adapter, plus persistence-across-reopen and sequence-durability tests for the filesystem adapter. That suite is the practical meaning of « swap storage without semantic change », and the yardstick any third-party adapter (including the paid ready-made ones) is validated against.

Host-Provided Storage

Chapter XII — Host-Provided Storage

The extension mechanism of the base product: a non-Rust host implements the storage in **its** language — its ORM, its connection pool, its database — and hands it to the Platform through one C callback. The Platform keeps every semantic (versioning, registry, rendering, pagination); the host supplies raw reads and writes over a small JSON protocol. This chapter documents the ABI, the operation protocol, buffer ownership, and the Python SDK's high-level wrapper.

12.1 — The ABI

Three additions to the C surface of chapter 13:

```
typedef int32_t (*poo2l_storage_cb)(void* user_data,
                                     const char* op_json,
                                     char** out_json);

char* poo2l_buf_alloc(uintptr_t len);           // allocate the reply buffer
void* poo2l_platform_new_hosted(poo2l_storage_cb cb, void* user_data);
```

`poo2l_platform_new_hosted` builds a Platform whose **entire storage** is delegated to `cb` (authorization stays allow-all, events no-op, sequences per-process — like the built-in configs). The returned handle is used with the ordinary `poo2l_platform_call` / `poo2l_platform_free`; `user_data` is passed back verbatim on every invocation and must outlive the handle.

Concurrency contract: callback invocations are serialized behind a mutex — the host callback is never invoked concurrently. This is deliberately Python-friendly (ctypes + GIL) and makes a plain database connection safe to use inside the callback.

12.2 — The wire protocol

Each storage operation is one callback invocation. The Platform passes a NUL-terminated operation document:

```
{ "op": "prompt.get", "tenant": "acme", "args": { "uid": "3f2a..." } }
```

The host replies by writing a NUL-terminated JSON document — `{"ok": <payload>}` on success, `{"error": {...}}` on failure — into a buffer obtained from `poo2l_buf_alloc`, stores the pointer in `*out_json`, and returns `0`.

Buffer ownership is one-directional: the Platform owns the allocator. The host allocates the reply with `poo2l_buf_alloc(len)` (allocate the byte length *including* the trailing NUL), fills it, hands it over — and never frees it; the Platform frees every reply buffer, using an internal length prefix to do so with its own allocator regardless of the host language. Any nonzero return code, missing buffer or malformed reply surfaces to the caller as a typed `storage` error naming the operation.

12.3 — The operations

Fifteen operations cover the four storage traits (`args` → `ok` payload):

op	args	ok payload
<code>prompt.get</code>	<code>{uid}</code>	the prompt object, or <code>null</code>
<code>prompt.put</code>	<code>{prompt}</code>	anything (ignored)
<code>prompt.delete</code>	<code>{uid}</code>	<code>true / false</code> — existed
<code>prompt.scan</code>	<code>{}</code>	array of prompt objects, stable order
<code>container.get</code>	<code>{path}</code>	the meta object, or <code>null</code>
<code>container.put</code>	<code>{path, meta}</code>	anything
<code>container.delete</code>	<code>{path}</code>	<code>true / false</code>
<code>container.list</code>	<code>{}</code>	object <code>{path: meta}</code>
<code>version.append</code>	<code>{uid, entry, content}</code>	anything
<code>version.list</code>	<code>{uid}</code>	array of entries, oldest first
<code>version.content</code>	<code>{uid, version}</code>	the snapshot string, or <code>null</code>
<code>version.purge</code>	<code>{uid}</code>	anything
<code>blob.put</code>	<code>{base64}</code>	the ref string
<code>blob.get</code>	<code>{ref}</code>	<code>{base64}</code> , or <code>null</code>
<code>blob.release</code>	<code>{ref}</code>	anything

The object shapes are the serde forms from appendix B — what `prompt.put` hands you is exactly what `prompt.get` must give back.

Two design points. First, **`prompt.list` is deliberately not delegated**: the Platform computes pagination and filtering in Rust over your `prompt.scan`, using the same reference implementation as the shipped adapters — so the trickiest semantics (cursor stability, field selection, prefix matching) are guaranteed identical no matter who stores the

data. Your side only has to return all templates of a tenant in a stable order. Second, the payload contract is checked: a reply that does not parse into the expected shape is a `storage` error naming the operation and the mismatch — fail loud, not corrupt.

12.4 — The Python wrapper

The SDK hides the FFI entirely. Subclass `HostStorage`, implement the operations as methods (dots become underscores: `prompt.get` → `template_get(tenant, args)`), return the payload, raise to signal a storage error:

```
from poo2l_platform import Poo2lPlatform, HostStorage

class PostgresStorage(HostStorage):
    def __init__(self, conn):
        self.conn = conn

    def template_get(self, tenant, args):
        row = self.conn.execute(
            "SELECT doc FROM templates WHERE tenant=%s AND uid=%s",
            (tenant, args["uid"])).fetchone()
        return row[0] if row else None

    def template_put(self, tenant, args):
        p = args["prompt"]
        self.conn.execute("INSERT INTO templates ... ON CONFLICT ... UPDATE ...",
                           (tenant, p["uid"], json.dumps(p)))
        return True

    def template_scan(self, tenant, args):
        return [r[0] for r in self.conn.execute(
            "SELECT doc FROM templates WHERE tenant=%s ORDER BY created, uid",
            (tenant,))]
    # containers / versions / blobs likewise...

platform = Poo2lPlatform(host_storage=PostgresStorage(conn))
```

The wrapper builds the C trampoline, allocates reply buffers correctly, serializes JSON both ways, converts raised exceptions into `{"error": ...}` replies, and — crucially — never lets a Python exception cross the FFI. An unimplemented operation raises `NotImplementedError`, which surfaces as a storage error the first time the Platform needs it: implement incrementally, the failure mode is loud and named.

`DictHostStorage` is the complete in-memory reference implementation — a starting point where each dict lookup marks the spot for your query. The runnable proof is `sdk/examples/host_storage_demo.py`: collections, includes, versioning and pagination, all flowing through host-side Python dicts.

12.5 — Semantics and caveats

Everything above the storage line behaves exactly as in the rest of this book: versioning rules, registry construction, the cache (invalidated by Platform mutations — if some *other* process writes to your database, call `invalidate`), events, rendering. Three operational caveats:

- **Stable scan order is your contract.** Sort by a creation timestamp then uid (or any stable equivalent) — registry construction and cursors depend on it.
- **Sequences are per-process** in hosted mode today (there is no `seq.*` operation); incremental indexers should re-scan after a restart, or the embedding can be revisited when the SPI grows a hosted SeqStore.
- **Latency multiplies.** Each operation is an FFI round-trip plus your query; the registry cache keeps render paths from hammering storage, but a cold build is one `prompt.scan` + one `container.list` — keep those two fast.

Ready-made database adapters (PostgreSQL, MySQL, ...) — native Rust, no callback overhead, maintained — are paid add-ons; the callback mechanism itself is, and stays, part of the base product.

Partie 5

Integration Surfaces

The C ABI and Language SDKs

Chapter XIII — The C ABI and Language SDKs

Every language binding of the Platform rests on one stable C ABI: a handful of exported functions, JSON in, JSON out. The bindings are thin and logic-free by design — a wrapper that contains no behavior cannot drift from the reference. This chapter documents the ABI precisely, then each shipped SDK (Python, Node, .NET, Java) and the packaging pipeline.

13.1 — The exported functions

```
void* poo2l_platform_new(const char* config_json);      // NULL on error
char* poo2l_platform_call(void* h, const char* method,
                          const char* tenant, const char* args_json);
void  poo2l_string_free(char* s);
void  poo2l_platform_free(void* h);

// Host-storage extension (chapter 12):
char* poo2l_buf_alloc(uintptr_t len);
void* poo2l_platform_new_hosted(poo2l_storage_cb cb, void* user_data);
```

poo2l_platform_new takes the storage configuration as JSON and returns an opaque handle, or NULL on a bad configuration:

```
{ "storage": "memory" }
{ "storage": "filesystem", "path": "/data/poo2l" }
{ "storage": "sqlite",     "path": "/data/poo2l.db" }
```

A NULL or missing config defaults to memory; an unknown storage kind, or a missing `path` where one is required, yields NULL. The filesystem and SQLite configs come fully wired — durable sequences included.

poo2l_platform_call is the whole method surface: a method name (appendix A), a tenant string (NULL defaults to "default"), and the method's arguments as a JSON object (NULL defaults to `{}`). It *always* returns a JSON document — never NULL:

```
{ "ok": <result> }
{ "error": { "error": "not_found", "kind": 'template', "key": "NOPE" } }
```

The error object is the `serde` form of the typed enum (appendix C), engine diagnostic codes verbatim. Malformed calls (null handle, non-UTF-8 method, unknown method, missing required argument) come back as `invalid_input` errors rather than crashes; a panic in the dispatch is caught and surfaces as a `storage` error — the ABI boundary never unwinds into the host.

Memory rules — two, and only two: every string returned by `poo2l_platform_call` is freed with `poo2l_string_free`, and every handle from either constructor is freed with `poo2l_platform_free`. (Host-storage reply buffers follow chapter 12's separate rule: allocated by `poo2l_buf_alloc`, freed by the Platform.)

Binary data travels as base64 inside the JSON envelope (`put_blob {base64}`, `get_blob → {base64}`) — the one concession JSON transport demands.

The library builds as `cdylib` (shared library for FFI), `staticlib` and `rlib` from the `pool-platform-c-api` crate; threading is safe — the handle is `Send + Sync`, with per-tenant locking inside.

13.2 — One dispatch, verified once

Internally, `poo2l_platform_call` delegates to the public Rust function `dispatch_json(platform, method, tenant, args)` — the same function the HTTP server mounts (chapter 14). Method names, argument shapes and result shapes are therefore identical across C, every SDK and HTTP; appendix A documents them once. The dispatch is covered by the Rust test suite, including a C-level round-trip test exercising the actual exported symbols.

13.3 — Python — `poo2l_platform`

A pure-Python ctypes wrapper (no compiled Python extension). Library discovery, in order: `P002L_PLATFORM_LIB`, the packaged `_native/` directory, then the workspace build tree (`target/release`, then `target/debug`).

```
from poo2l_platform import Poo2lPlatform, PlatformError

with Poo2lPlatform({"storage": "sqlite", "path": "poo2l.db"}, tenant="acme") as p:
    rec = p.create_template({"name": "Hello", "content": "@@@name Hello\n\nHi @@@=USER"}
    out = p.render_template("HELLO", {"system": {"USER": "alice"}})
```

The surface mirrors the dispatch one-to-one — `create_template`, `update_template`, `patch_template_metadata`, the trash trio, `get_template`, `list_templates`, `versions`, `export_package/import_package`, `containers`, `render_template/render_source`, `validation/analysis`, `dependencies/candidates`, `build_registry`, `invalidate`, blobs (bytes in, bytes out — base64 handled for you) — plus three conveniences: `iter_templates(**query)` follows cursors for you; `call(method, args, tenant=...)` reaches any method generically (including per-call tenant override); and the context-manager/`__del__` pair frees the handle. Errors raise `PlatformError` with `kind` (the discriminant), `code` (engine diagnostics only) and `data` (the full object). The host-storage entry point (`host_storage=`) is chapter 12.

13.4 — Node — @poo2l/platform

`sdk/node/index.js`, built on `koffi` (FFI without `node-gyp`; the dependency loads lazily so the package can be inspected without it). Same discovery order (with `native/` as the bundled directory), same envelope, camelCase method names:

```
const { Poo2lPlatform } = require('@poo2l/platform');
const platform = new Poo2lPlatform({ storage: 'sqlite', path: 'poo2l.db' }, { tenant:
const out = platform.renderTemplate('HELLO', { system: { USER: 'alice' } }));
```

Errors throw `PlatformError` with the same `kind/code/data` triple. Host-storage exposure in Node is on the roadmap (the mechanism is language-agnostic; only the trampoline is per-language).

13.5 — .NET and Java — reference wrappers

Both consume the same four functions and expose the generic `Call(method, tenant, argsJson)` shape, leaving idiomatic sugar to the application:

- **.NET** (`sdk/dotnet/Poo2lPlatform.cs`): `P/Invoke` against `pool_platform_c_api`; `PlatformException` carries the error as a `JsonElement` with `Kind/Code` accessors; `IDisposable` frees the handle. Ship the native library as a NuGet runtime asset (`runtimes/<rid>/native`).
- **Java** (`sdk/java/Poo2lPlatform.java`): `java.lang.foreign` (Panama, JDK 22+; a JNA variant is a mechanical translation); `AutoCloseable`; `P002L_PLATFORM_LIB` or the system library path locates the native library. Ship it as a JAR resource extracted at startup.

13.6 — Packaging and the demos

`sdk/package.sh` assembles distributables for the OS it runs on: `release-builds` the C ABI, copies the library into `sdk/python/poo2l_platform/_native/` (building a wheel when `python -m build` is available) and into `sdk/node/native/`. Cross-OS artifacts are CI's job — the multi-OS pipeline builds and gates on Linux, macOS and Windows (appendix D).

Two runnable, self-asserting examples double as living documentation and CI gates: `sdk/examples/demo_backend.py` — a complete vendor integration: `provision` (collections, inheritance, includes, parameters), `render` with host-computed system variables, `version` and `retrieve` history, typed-error handling, `enumerate` with field selection and paging, full-fidelity export including metadata bags, blobs; and `sdk/examples/host_storage_demo.py` — the same Platform running entirely on host-side Python storage (chapter 12).

The HTTP Server

Chapter XIV — The HTTP Server

The Platform is embeddable first; the server exists for the cases embedding cannot cover — environments that will not load a native library, process isolation requirements, independent update cycles, containers. It is deliberately *not a second implementation*: one generic endpoint exposes the exact same `dispatch_json` the C ABI uses, so everything in this book — methods, argument shapes, results, errors — applies unchanged. What the server adds is transport: routing, status-code mapping, configuration, a health check and a container image.

14.1 — The surface

Two routes :

```
POST /v1/{tenant}/{method}    body = the method's arguments (JSON object)
GET  /health
```

The method segment is any dispatch method of appendix A; the body may be omitted entirely (it defaults to `{}` — convenient for `list_containers` or `build_registry`). Success is `200` with `{"ok": ...}`; failure carries the typed error object under `{"error": ...}` and a mapped status:

error discriminant	HTTP status
<code>not_found</code>	404
<code>conflict</code>	409
<code>invalid_input</code> , <code>cyclic_reference</code>	400
<code>permission_denied</code>	403
<code>engine_diagnostic</code>	422
anything else (<code>storage</code> , <code>serialization</code>)	500

The body is authoritative; the status is a convenience for HTTP middleware. Note `422` for engine diagnostics: a source that fails to render is not a malformed request — it is a semantically unprocessable document, and the full diagnostic (code verbatim, message, handle, phase) is in the body.

`GET /health` returns liveness plus the embedded engine's conformance level:

```
{ "status": "ok", "service": "poo2l-platform-server", "engine_implemented_level": 5 }
```

Rendering is CPU-bound, so the server executes each dispatch on a blocking-safe path (`block_in_place`), keeping the async runtime responsive under concurrent renders.

14.2 — Configuration

Environment variables, each with a flag override :

Env	Flag	Meaning	Default
P002L_STORAGE	<code>--storage</code>	<code>memory</code> <code>filesystem</code> <code>sqlite</code>	<code>memory</code>
P002L_PATH	<code>--path</code>	data path (required for filesystem/sqlite)	—
P002L_ADDR	<code>--addr</code>	bind address	<code>127.0.0.1:7433</code>

```
pool-platform-server --storage sqlite --path /data/poo2l.db --addr 0.0.0.0:7433
```

A bad configuration (unknown storage, missing path, unbindable address) exits with code 2 and a message on stderr ; shutdown is graceful on Ctrl-C.

14.3 — No authentication, by design

The server ships without authentication — the plan's principle : *the host owns users and sessions*. The `{tenant}` path segment is trusted as given. Deployment models that follow : bind to localhost and let the co-located application (which did authenticate its user) choose the tenant per request ; or put a reverse proxy in front that authenticates and pins/validates the tenant segment ; or skip the server and embed. What is *not* a supported model is exposing the raw server to untrusted callers — anyone who can reach it can name any tenant.

14.4 — The container image

A two-stage Dockerfile ships next to the crate (build context = repository root, so the engine crate is visible to the build) :

```
cd src/platform
docker build -f crates/pool-platform-server/Dockerfile -t poo2l-platform-server ../..
docker run -p 7433:7433 -v poo2l-data:/data poo2l-platform-server
```

The image defaults to SQLite at `/data/poo2l.db` (declared `VOLUME`), binds `0.0.0.0:7433` , exposes the port and wires a `HEALTHCHECK` against `/health` . Same warning as above : publish the port to trusted networks only, or front it with an authenticating proxy.

14.5 — Choosing between server and embedding

Embedding wins on latency (no HTTP hop), events (in-process `EventSink`), and host-storage callbacks (C-ABI-only today). The server wins on isolation, polyglot fleets sharing one store, independent upgrades, and environments where native libraries are unwelcome. Because both speak the same dispatch, the choice is reversible: code written against the method surface — which is to say, against appendix A — moves between them by swapping the transport line.

Appendices

Complete Method Index

Appendix A — Complete Method Index

The full method surface, valid identically across the C ABI (`poo2l_platform_call`), every SDK, and HTTP (`POST /v1/{tenant}/{method}`). Arguments are the JSON object passed as `args_json` /request body; results are the value under `"ok"` . Object shapes are in appendix B ; error shapes in appendix C. The chapter column points to the full discussion.

A.1 — Template lifecycle

Method	Arguments	Result	Ch.
<code>create_template</code>	<code>CreateTemplate</code> object (at top level)	<code>TemplateRecord</code>	4
<code>update_template</code>	<code>{uid, patch: UpdateTemplate}</code>	<code>TemplateRecord</code>	4
<code>patch_template_metadata</code>	<code>{uid, namespace, patch?} \ {uid, namespace, delete: true}</code>	<code>TemplateRecord</code>	4
<code>delete_template</code>	<code>{uid}</code>	<code>{done: true}</code>	4
<code>restore_template</code>	<code>{uid}</code>	<code>{done: true}</code>	4
<code>purge_template</code>	<code>{uid}</code>	<code>{done: true}</code>	4
<code>get_template</code>	<code>{key}</code> — <code>uid template_id alias</code>	<code>TemplateRecord</code>	4
<code>list_templates</code>	<code>ListQuery</code> object (at top level)	<code>Page<TemplateSummary></code>	9

A.2 — Versions

Method	Arguments	Result	Ch.
<code>list_versions</code>	<code>{uid}</code>	array of <code>VersionEntry</code> , oldest first	4
<code>get_version_content</code>	<code>{uid, version}</code>	the snapshot string	4

A.3 — Packages

Method	Arguments	Result	Ch.
ex-port_package	{include_deleted?: bool}	the poo2l_package: "1.0" envelope	10
im-port_package	{package: envelope, options?: {}}	ImportReport	10

A.4 — Containers

Method	Arguments	Result	Ch.
list_containers	{}	object {path: ContainerMeta}	5
put_container	{path, patch: ContainerPatch}	ContainerMeta	5
patch_container_metadata	{path, namespace, patch?} \\ {path, namespace, delete: true}	ContainerMeta	5
delete_container	{path, cascade?: "block" \\ "detach"}	ContainerDeletion	5
resolve_collection	{path}	array of TemplateSummary	5

A.5 — Language services

Method	Arguments	Result	Ch.
render_template	{key, request?: RenderRequest}	RenderOutcome	6
render_source	{source, request?: RenderRequest, as_template?: key}	RenderOutcome	6
validate_template	{key}	Validation	7
validate_source	{source}	Validation	7
analyze_template	{key}	Analysis	7
analyze_source	{source}	Analysis	7

Method	Arguments	Result	Ch.
<code>list_dependencies</code>	<code>{key}</code>	array of <code>DependencyRef</code>	7
<code>include_candidates</code>	<code>{key}</code>	array of <code>Candidate</code>	7

A.6 — Registry and blobs

Method	Arguments	Result	Ch.
<code>build_registry</code>	<code>{}</code>	<code>RegistrySnapshot</code> — <code>{entries, seq}</code>	8
<code>invalidate</code>	<code>{}</code>	<code>{done: true}</code>	8
<code>put_blob</code>	<code>{base64}</code>	the blob ref string (<code>sha256:...</code>)	9
<code>get_blob</code>	<code>{ref}</code>	<code>{base64}</code>	9

An unknown method, or a missing required argument, returns the `invalid_input` error shape — never a transport-level failure.

A.7 — Rust-only surface

Available when embedding `pool-platform-core` directly (not on the JSON dispatch):

API	Purpose	Ch.
<code>Platform::new(PlatformConfig) / Platform::in_memory()</code>	construction ; the config wires the seven SPI slots	2, 11
<code>scan_templates(&t, f)</code>	streaming full-fidelity scan with early stop	9
<code>render_with_registry(source, req, entries, caller_folder)</code>	render against an explicit registry, no store	6
<code>engine_implemented_level()</code>	the embedded engine's conformance level	7
<code>pool_platform_core::engine::*</code>	the raw <code>pool-core</code> API, re-exported verbatim	7
EventSink wiring (<code>PlatformConfig.events</code>)	in-process business events	9
<code>query::paginate(all, query)</code>	the reference list semantics, as a helper for adapter authors	11

Constructors on the C ABI side — `poo21_platform_new(config_json)`, `poo21_platform_new_hosted(cb, user_data)`, plus `poo21_string_free`, `poo21_platform_free` and `poo21_buf_alloc` — are chapter 13 (and 12 for the hosted pair).

Data Model Reference

Appendix B — Data Model Reference

The JSON shapes of every object crossing the API, exactly as serialized. Conventions: timestamps are RFC3339 UTC strings with second precision (`2026-07-12T22:01:37Z`); optional-and-absent fields are omitted where noted ; enum values are snake_case strings.

B.1 — StoredTemplate / TemplateRecord

`TemplateRecord` = the `StoredTemplate` fields flattened, plus `version` :

```
{
  "uid": "3f2a9c...",
  "template_id": "CODE_REVIEW",
  "name": "Code review",
  "content": "@@@name Code review\n...",
  "description": "",
  "collection": "Team/Reviews",
  "inherits": null,
  "virtual": false,
  "disabled": false,
  "deleted": false,
  "deleted_at": null,
  "aliases": [],
  "deprecated": null,
  "diamond_policy": "default",
  "created": "2026-07-12T22:01:37Z",
  "modified": "2026-07-12T22:14:02Z",
  "metadata": { "x_myapp": { "tags": ["review"] } },
  "version": "0.3"
}
```

`diamond_policy` ∈ `default` | `merge` | `refuse`. `metadata` is any JSON value (`null` when never set). `version` is `null` when no history exists.

B.2 — CreateTemplate

All fields optional except that at least one of `name` / `template_id` must be non-blank: `name`, `content`, `template_id`, `description`, `collection`, `inherits`, `virtual`, `disabled`, `aliases`, `diamond_policy`, `metadata`.

B.3 — UpdateTemplate

Partial patch — omitted fields unchanged: `name`, `content`, `template_id`, `description`, `virtual`, `disabled`, `aliases`, `diamond_policy`, `metadata`, plus the double-option fields `collection`, `inherits`, `deprecated` (explicit `null` clears; omission leaves unchanged) and the riders `major_version` (bool), `version_note` (string), `restore_operation` (bool).

B.4 — MetadataBlockPatch

```
{ "namespace": "x_myapp", "patch": { "pinned": true, "old": null } }
{ "namespace": "x_myapp", "delete": true }
```

Exactly one of `patch` (RFC 7386 merge-patch) / `delete`.

B.5 — TemplateSummary

```
{
  "uid": "3f2a9c...", "template_id": "CODE_REVIEW", "name": "Code review",
  "description": "", "collection": "Team/Reviews", "inherits": null,
  "virtual": false, "disabled": false, "deleted": false,
  "created": "...", "modified": "...",
  "content": "...",           // only when fields = with_content | full
  "metadata": { ... }        // only when fields = full
}
```

`content` / `metadata` are *absent* (not null) unless selected.

B.6 — ListQuery / Page

```
{ "collection_prefix": "Team", "include_deleted": false, "uids": [],
  "modified_since": "2026-07-01T00:00:00Z",
  "fields": "summary" | "with_content" | "full",
  "cursor": "12", "limit": 100 }
```

Result page: { `"items": [TemplateSummary...]`, `"next_cursor": "37" | absent` }. `limit` 0/absent = 500. Cursors are opaque.

B.7 — VersionEntry

```
{ "version": "1.0", "operation": "create" | "update" | "restore" | "migrated",
  "datetime": "2026-07-12T22:01:37Z", "note": "big rewrite" | null,
  "checksum": "a1b2c3d4" }
```

`checksum` = first 8 hex chars of SHA-256 over line-ending-normalized content.

B.8 — ContainerMeta / ContainerPatch / ContainerDeletion

```
{ "kind": "folder" | "collection" | "library" | "project",
  "description": "", "label": null,
  "inherits": null,           // collections only – a container PATH
  "requires": [],            // libraries only – UPPERCASE names
  "metadata": null }
```

`ContainerPatch`: same fields, all optional; `label/inherits` double-option.
`ContainerDeletion`: { `"trashed_uids": [...]`, `"removed_paths": [...]`,
`"detached_inherits": [...]` }.

B.9 — RenderRequest / FrameRequest

```
{
  "parameters": { "language": "rust" },
  "system": { "USER": "alice", "DATE": "2026-07-12" },
  "section_toggles": { "examples": false },
  "constants": { "tone": "formal" },
  "attachments": { "logs": "...runtime content..." },
  "parameter_set": "strict",
  "include_depth": 8,
  "frames": { "review.inner": { "parameters": { "n": "3" },
                                "section_toggles": {}, "parameter_set": null } }
}
```

Reserved system names (routed to the engine's system context, explicit parameters overriding host values, constants filling gaps only — chapter 6): `DATE` `TIME` `DATETIME` `YEAR` `MONTH` `DAY` `WEEKDAY` `USER` `USERNAME` `LANG` `GROUPS` `GROUPS_ADMIN`.

B.10 — RenderOutcome

```
{ "text": "...",
  "warnings": [ { "code": "...engine code...", "message": "...", "handle": "X" | absent } ],
  "attachments": {
    "included": [ { "name": "spec", "type": "pdf", "kind": "included", "frame": "" },
    "to_include": [ { "name": "logs", "type": "text", "kind": "to_include", "frame": "" } ]
  }
```

The only non-engine warning code: `INVALID_REGISTRY_ENTRY` (chapter 6).

B.11 — Validation

```
{ "ok": true, "kind": 'template' | "collection" | "library" | "unstructured" | absent,
  "warnings": [ "..." ],
  "errors": [ { "code": "...", "message": "...", "phase": 1 | absent } ] }
```

B.12 — Analysis

```
{ "kind": 'template', "id": "Marketing.WELCOME" | absent, "name": "..." | absent,
  "description": "..." | absent, "inherits": "..." | absent,
  "virtual": false, "final": false,
  "params": [ { "name": "lang", "type": "text", "default": "fr" | absent,
    "select_options": [ "a", "b" ] | absent,
    "min": 0, "max": 10, "minlen": 1, "maxlen": 80, "pattern": "...", // eac
    "optional": false, "required": true, "label": "Language" | absent } ],
  "consts": [ { "name": "MAX", "type": "number", "value": "100" } ],
  "sections": [ { "name": "main", "description": "..." | absent,
    "togglable": false, "default_on": true,
    "virtual": false, "final": false,
    "delete": false, "transparent": false,
    "placement": { "kind": "after", "anchor": "intro" } | absent } ],
  "includes": [ { "handle": "Common.DISCLAIMER",
    "args": [ ["lang", "fr"] ], "transparent": false,
    "has_args": true, "section_after": ["intro", "one"],
    "candidates": ["Common.DISCLAIMER"] } ],
  "warnings": [ "..." ] }
```

B.13 — DependencyRef / Candidate / RegistrySnapshot

```
{ "handle": "PARENT", "relation": "inherits" | "include", "resolved": true }
{ "handle": "Common.DISCLAIMER", "aliases": ["DISCLAIMER"],
  "kind": 'template' | "collection", "target": "Common.DISCLAIMER" }
{ "entries": [ ["KEY", "...source...", ... ], "seq": 42 }
```

B.14 — PlatformEvent

```
{ "tenant": "acme", "seq": 42, "type": "template_updated", "uid": "3f2a..." }
```

`type` ∈ `template_created` | `template_updated` | `template_deleted` | `template_restored` | `template_purged` | `container_changed` | `container_deleted` | `version_appended` | `blob_stored` ; payload key is `uid`, `path`, `uid + version`, or `blob` per type (chapter 9).

B.15 — Package envelope / ImportReport

Envelope: { "poo2l_package": "1.0", "exported_at": ..., "templates": [...],
"collections": [...], "libraries": [...], "external_dependencies": [{ "kind":
'template' | "collection" | "library", "reference": "..." }],
"x_poo2l_platform": { "folders": { path: { "kind", "label"?,
"description"?, "metadata"? } } } } . Record shapes: chapter 10. ImportReport:

```
{ "templates_created": 12, "templates_skipped": 1,  
  "containers_created": 3, "containers_skipped": 0, "warnings": [ "..." ] }
```

B.16 — Blob references

Strings of the form `sha256:<64 hex chars>` (shipped adapters; host storage may mint its own scheme). Over JSON transport, blob bytes travel as `{ "base64": "..." }`.

Error Reference

Appendix C — Error Reference

One typed enum crosses every surface. In Rust it is `PlatformError` ; over JSON it serializes with an internal `"error"` tag ; over HTTP it additionally maps to a status code. Engine diagnostics are transported verbatim — never reinterpreted, never re-worded.

C.1 — The variants

error	Fields	Meaning	HTTP
<code>not_found</code>	<code>kind</code> , <code>key</code>	No such entity. <code>kind</code> $\in$ <code>prompt</code> <code> </code> <code>container</code> <code> </code> <code>version</code> <code> </code> <code>blob</code> <code> </code> <code>tenant</code> .	404
<code>conflict</code>	<code>kind</code> , <code>key</code> , <code>reason</code>	The operation contradicts current state : duplicate <code>template_id</code> in a collection, rendering a disabled prompt, container deletion blocked by inheriting descendants.	409
<code>invalid_input</code>	<code>field</code> , <code>reason</code>	A malformed argument : empty name, major bump without note, bad cursor, invalid nesting, unknown method, missing required argument.	400
<code>permission_denied</code>	<code>action</code> , <code>key</code>	An <code>AuthorizationProvider</code> said no.	403
<code>engine_diagnostic</code>	<code>code</code> , <code>message</code> , <code>handle?</code> , <code>phase?</code>	A blocking engine diagnostic — see below.	422
<code>cyclic_reference</code>	<code>kind</code> , <code>path</code> (array)	A staged cycle : collection inheritance or library requires ; <code>path</code> is the offending chain.	400
<code>storage</code>	<code>detail</code>	An adapter or host-storage failure (I/O, SQL, callback error, internal panic at the ABI boundary).	500
<code>serialization</code>	<code>detail</code>	A (de)serialization failure in an adapter.	500

JSON examples :

```
{ "error": "not_found", "kind": 'template', "key": "NOPE" }
{ "error": "conflict", "kind": 'template', "key": "WELCOME",
  "reason": "a live prompt with this id already exists in this collection" }
{ "error": "invalid_input", "field": "version_note", "reason": "a major version requi
{ "error": "engine_diagnostic", "code": "UNDEFINED_REFERENCE",
  "message": "undefined reference `MISSING`", "handle": "MISSING", "phase": 3 }
{ "error": "cyclic_reference", "kind": "container", "path": ["A", "B", "A"] }
```

The C ABI wraps these under `{"error": {...}}`; SDK exceptions expose the discriminant as `kind` and, for engine diagnostics, the engine code as `code`.

C.2 — Engine diagnostics

When the engine raises a blocking diagnostic, the Platform keeps the *earliest blocking* one (the conformance identity) and transports it as `engine_diagnostic`:

- **code** — the engine's SCREAMING_SNAKE spec code, verbatim: `UNDEFINED_REFERENCE`, `AMBIGUOUS_INCLUDE`, `DUPLICATE_DECLARATION`, `UNIMPLEMENTED_VIRTUAL_TEMPLATE`, `SYSTEM_VARIABLE_OVERRIDE`, ... The authoritative catalogue is the language standard (§6.2); the Platform adds none and removes none.
- **message** — the engine's full message string.
- **handle** — best-effort extraction of the offending identifier from the message (first backquoted token, else the first ALL_CAPS dotted token). A convenience for UI highlighting; may be absent.
- **phase** — the engine pipeline phase, when the spec pins one (1 = parsing, etc.).

Non-blocking engine diagnostics are **not errors**: they arrive as `warnings` on successful results (render outcomes, validations, analyses) — code and message verbatim, plus the Platform-synthesized `INVALID_REGISTRY_ENTRY` described in chapter 6.

C.3 — Stable reason codes

Some `invalid_input` / `conflict` reasons embed stable UPPER_SNAKE codes hosts can match on, defined by the container rules:

Code	Raised by
<code>INVALID_NESTING</code>	<code>put_container</code> — kind not allowed under an ancestor
<code>INHERITS_ON_NON_COLLECTION</code>	<code>put_container</code> — inheritance on/to a non-collection
<code>INVALID_LIBRARY_NAME</code>	<code>put_container</code> — library leaf not <code>[A-Za-z][A-Za-z0-9_]*</code>
<code>CIRCULAR_LIBRARY_DEPENDENCY</code>	<code>put_container</code> — cycle in staged <code>requires</code>

Code	Raised by
INHE- RITS_DESCEN- DANTS_BLOCK_DELETE	<code>delete_container</code> in <code>block</code> mode — inheriting descendants listed after the colon

(Direct cycles also surface as the structured `cyclic_reference` variant; the codes above cover the string-reason cases.)

C.4 — Error discipline for integrators

Three properties to build on. **Failed mutations do not mutate**: validation precedes writes (the major-note rule is checked before anything is stored; container validation before the row is written). **The ABI never unwinds**: panics are caught at the boundary and surface as `storage` errors; malformed calls surface as `invalid_input` — a binding can always parse what comes back. **Warnings are data, errors are types**: anything non-blocking arrives in-band on success; anything blocking is one of the eight variants above — there is no third channel.

Guarantees, Gates and Proofs

Appendix D — Guarantees, Gates and Proofs

The Platform’s central claims are enforced mechanically, not aspirationally. This appendix collects each guarantee, what enforces it, and where the proof lives — the checklist to re-run after any change, and the yardstick for third-party extensions.

D.1 — Governance : the engine is untouchable

The single non-negotiable rule (`GOVERNANCE.md`) : **no file under the engine tree is ever modified by Platform work** — not the crates, not the workspace manifest, not the scripts. The Platform consumes only the existing public API of `pool-core` , as a plain Rust path dependency, from its own *sibling* Cargo workspace (joining the engine workspace would itself require editing an engine file). If a Platform need ever seems to require an engine change, the need is written down and submitted to the author ; work on that point suspends until the author decides — no unilateral action.

Enforcement is two-fold : the engine conformance suite must stay green and identical before and after any Platform commit, and CI’s *engine immutability gate* fails any run where a tracked engine file differs.

D.2 — Conformance parity — 511 cases through the Platform

The engine proves itself against the POO²L standard-conformance pack : 511 cases (501 pass, 10 documented gaps run as xfail), each with a source, a registry, a context, and either a byte-exact expected output or an expected error code (with phase, when pinned).

The Platform replays the **entire pack through its own render path** : pack registries are loaded as stored templates, contexts become `RenderRequest` s (system variables, parameters, toggles, parameter sets, frames, include-depth), and outputs are compared byte-for-byte — expected errors compared by code and phase. Zero divergence is required. This is the strongest statement in the book : storage → registry construction → context routing → engine, *the full Platform stack*, is grammar-equivalent to the reference runner on every standard case.

Proof: the `conformance_parity` integration test of the core crate, run in CI on every platform change, on three operating systems.

D.3 — Adapter equivalence

One behavioral suite — CRUD with version arithmetic, containers with collection-qualified rendering, listing filters, container deletion with undo info, blobs, purge — runs identically against the filesystem and SQLite adapters (the memory adapter backs the

unit tests of the same behaviors). Persistence is separately proven: a store reopened by a fresh Platform instance serves the same records, versions and renders; event sequences continue rather than restart (the durable `SeqStore`).

Host-provided storage inherits the equivalence a different way: pagination and filtering are *computed in Rust* over the host's scan by the same reference implementation the shipped adapters follow, and the hosted channel is exercised end-to-end by an in-repo host simulation plus the Python demo.

Proof: the storage crate's adapter tests; the c-api crate's hosted tests; `sdk/examples/host_storage_demo.py`.

D.4 — Fidelity and events

- **Lossless reconstruction:** `scan` + `list_containers` + `list_versions` + `get_blob` reconstruct a tenant without loss — metadata bags verbatim, foreign `x_*` blocks untouched. The package exporter (chapter 10) is built on exactly this surface, and the integration suite asserts full-fidelity enumeration including bags.
- **Exactly one event per mutation, strictly increasing per-tenant sequence**, emitted after commit; durable across restarts on filesystem and SQLite (best-effort persistence that never fails the committed mutation).

Proof: the events/cache integration tests; the sequence-durability adapter test.

D.5 — Performance

The acceptance criterion is comparative, not absolute: the same corpus (a shared collection member included by a hundred templates, rendered hundreds of times with parameters and system variables) runs through the historical Python path — the oracle — and through the Platform's C ABI. The benchmark asserts **byte-identical text** on every render and requires the Platform's throughput to be equal or better. The measured result at the time of writing: the Platform at 0.84× the oracle's wall-clock (faster), with identical bytes.

Proof: `benchmarks/bench_render.py` (run from the repository root with both paths on `PYTHONPATH`).

D.6 — The CI pipeline

Every change under the platform, the engine or the conformance pack runs four gates on a Linux/macOS/Windows matrix (`platform-ci.yml`):

1. **Engine conformance** — the canonical runner: 501 pass, 0 fail, 10 documented xfail.
2. **Platform tests** — the whole workspace, conformance parity included.
3. **Engine immutability** — `git diff` must be clean under the engine tree.
4. **SDK demos** — the vendor-backend and host-storage examples run against the freshly built C ABI through the Python SDK.

One portability detail encoded in the workflow : git line-ending translation is disabled before checkout, because the pack's expected outputs are compared byte-exact and CRLF rewriting on Windows would corrupt them.

D.7 — The local checklist

The same gates, runnable from a working copy (also listed in the workspace README) :

```
cd src/platform
(cd ../engine && bash conformance.sh)          # 1. engine: same outcome set as the refer
cargo test --workspace                        # 2. platform tests + parity
git diff --exit-code -- ../engine             # 3. engine untouched
cargo build --release -p pool-platform-c-api
(cd ../../ && python3 src/platform/sdk/examples/demo_backend.py)      # 4a
(cd ../../ && python3 src/platform/sdk/examples/host_storage_demo.py)  # 4b
(cd ../../ && PYTHONPATH=src:src/platform/sdk/python \
    python3 src/platform/benchmarks/bench_render.py)                # 5. perf
```

D.8 — Versioning of the Platform itself

The workspace ships as version `0.1.0`, proprietary license, `publish = false` — pre-1.0 by design : the public contract (`PLATFORM-API.md`) is a draft until the 1.0 freeze, which reviews it against real integration use before locking it. The package format carries its own version (`poo2l_package: "1.0"`, standard-defined) independent of the crate version. Forward plan — host-data namespaces, native SQL filtering, paid ready-made adapters, the one-shot application migration — lives in `ROADMAP.md`, decision history in the plan's questions journal.

D.9 — Continuity : what survives the vendor

Two commitments outlive the vendor, and both are written into binding text rather than promised in a manual.

The **language** relicenses from CC BY-ND 4.0 to CC BY 4.0 on a continuity event — dissolution without a successor rights-holder, or twenty-four months of silence — so a frozen specification can be corrected and extended by whoever is left (`standard-governance.md` 7.5).

The **product** licence (EULA article 10, `downloads/LICENSE.txt`) acts in two steps, because continuing to work and owning the sources are not the same need on the same clock. From the moment of the event the licence stops being revocable and use continues uninterrupted, with **no activation or revalidation** required — no third-party outage can end it. Only if no transferee is found within six months do the **sources pass to the Apache License 2.0**.

That second step is a **grant, not a delivery**, and the distinction is the whole reason it works : a grant takes effect by the contract itself, with no act from the vendor, in the branch where there is no vendor left to act. It carries the matching limit, stated plainly in the clause — anyone lawfully holding a copy of the sources may from then on use, modi-

fy, publish and redistribute them, but the clause obliges nobody to hand a copy over. It covers the sources and nothing else : the POO²L specification runs on its own licence and its own clause, third-party components keep theirs, and the trademark is expressly outside an Apache grant.

The trigger is deliberately not insolvency alone : documented abandonment — no update and no support answer for twelve months, after a ninety-day formal notice — fires while the clause is still plainly enforceable. That twelve-month clock governs the product ; the specification's twenty-four-month clock governs the text. Separate assets, separate periods.

Neither commitment is restated here, and this appendix is not the contract : the binding text governs, and if this page ever disagrees with it, the contract is right.

Proof: `tests/test_legal_texts.js` — the clause is present in both binding texts, the licence's own survival list names it (without which it would die with the licence), and the five shipped copies of the licence are byte-identical.

Resources

Télécharger

- Format PDF (10-ptatform-complete-guide_v1.3_en_20261008.pdf)
- Format EPUB (10-ptatform-complete-guide_v1.3_en_20261008.epub)
- Format Markdown (10-ptatform-complete-guide_v1.3_en_20261008.md)