

POO²L Engine API Guide

POO²L Engine API Guide

Calling the Engine from Rust, JavaScript and Python

InZeMobile

TABLE DES MATIÈRES

Partie 1 — The Contract

Chapter I — The Rendering Contract	7
Chapter II — The Inputs by Example	11

Partie 2 — The Bindings

Chapter III — The Rust Core API	19
Chapter IV — The JavaScript / WebAssembly API	22
Chapter V — The Python API	25

Partie 3 — Diagnostics

Chapter VI — Diagnostics: log and trace	31
---	----

Appendices

Appendix A — Context, Registry and Diagnostics Reference	37
Resources	45

Partie 1

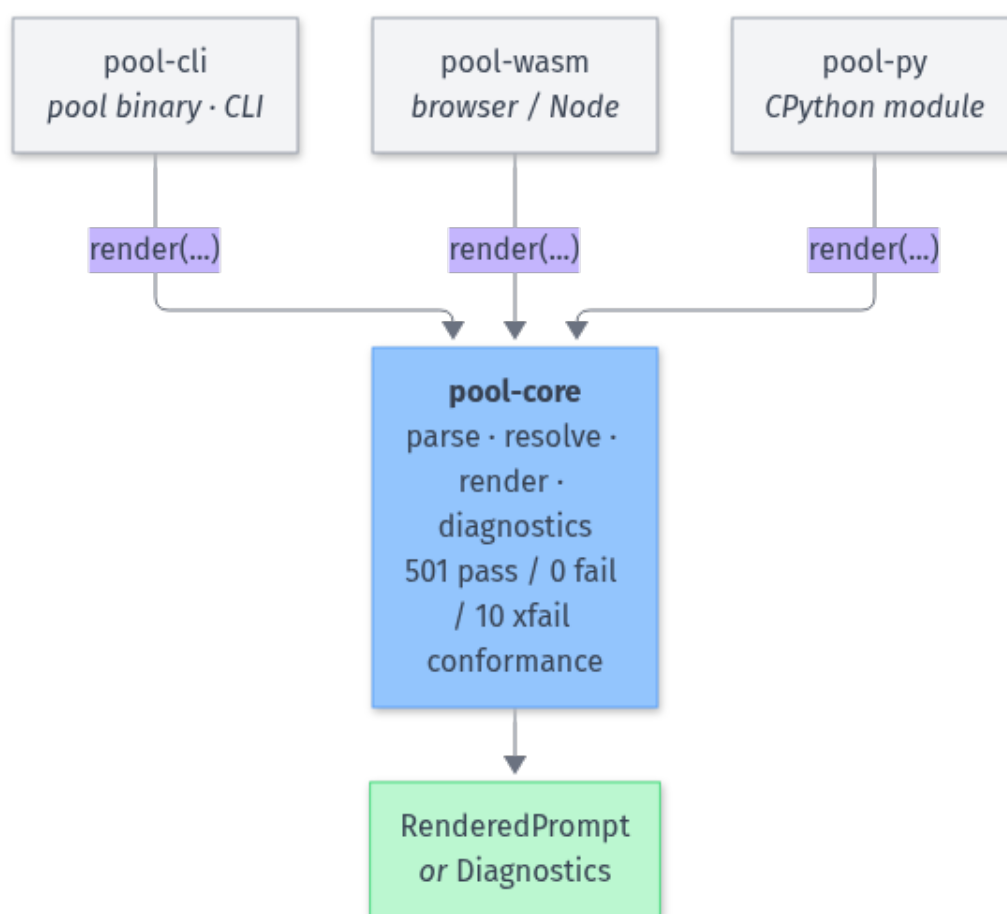
The Contract

Chapter I — The Rendering Contract

This guide is about *calling* the engine. Obtaining and installing it — building the `pool` tool, the WebAssembly package, and the Python wheel — is the subject of the companion **installation guide**. Here we assume the engine is in place, and describe its programming interface.

Every binding exposes the same contract, because they are thin shims over the same core. Learn it once here; the per-language chapters that follow show how it looks in Rust, JavaScript and Python.

One implementation, not three. *The bindings add no language logic. All parsing, resolution, rendering and diagnostics live exactly once, in the core (`pool-core`); each binding is a thin marshalling layer (~304 lines for Python, ~353 for WebAssembly) that only moves the host's inputs into the call and the result back. So the three APIs are not reimplementations of one another — they are the same code. Two consequences matter to you: every binding returns **byte-identical** output for identical input, and the single conformance pack that validates the core validates them all. There is no second implementation to drift out of sync.*



1.1 — The central function

The heart of the API is a single call :

```
render(source, context, registry) → RenderedTemplate | Diagnostics
```

It takes three inputs and returns one of two outcomes. There is no setup, no session, no hidden state : the same inputs always produce the same output, and the call has no side effects.

`render` does not stand alone, though. The public API is a small, deliberately narrow surface of four functions — this is what the bindings expose and what IziBuddy consumes in production :

Function	Purpose	Returns
<code>render</code>	instantiate a prompt to final text	rendered prompt, or a blocking diagnostic
<code>validate</code>	parse-check a source, no render, no registry	{ ok, kind, warnings, errors }
<code>analyze</code>	extract static structure, no render	{ ok, kind, params, sections, includes, ... }
<code>scope_candidates</code>	list <code>@@@include</code> targets in scope (editor autocomplete)	[{ handle, kind }]

`render_with` (chapter 6) is a fifth entry that adds the diagnostics flag string ; `render(...)` is exactly `render_with(..., "")`.

`validate(source)` is the « does it parse ? » gate for editors and migration tooling : it runs the parsing phases only — no registry, no include resolution, no render — and returns the document `kind` plus advisory `warnings`, or the `errors` it found (a prompt with no `@@@description` reports `MISSING_DESCRIPTION` in `warnings`). **`analyze(source)`** goes one step further on the *static* structure : it returns the declared `params`, `consts`, `sections` and `includes` (still without resolving or rendering anything) — the engine-backed replacement for hand-rolled grammar analysis. **`scope_candidates`** builds the world from `source` + `registry` and lists the include handles reachable from a prompt's folder, for autocomplete. `validate` and `analyze` are exposed by every binding ; `scope_candidates` is exposed by the Rust core and the Python binding. The per-language chapters and the reference appendix give the exact shapes.

1.2 — The three inputs

Source — the `.pool` text to instantiate.

Context — the instantiation inputs for the root frame : parameter values, system variables, section toggles, a selected parameter set, attachment bindings, an include-depth limit, per-frame inputs, the folder the entry prompt lives in. Every field is optional ; an empty context renders a prompt with its declared defaults.

Registry — the entities the source may `@@@include` or inherit from. This is the engine's **only** contact with the outside world, and it is *injected* : the core never reads a file, a database or the network itself. The host provides the available entities ; the core parses and resolves them. (See the reference appendix for the exact shape of the context and registry.)

1.3 — The two outcomes

Success — a `RenderedTemplate` with three parts :

- `text` — the assembled prompt, byte-exact, with its structure carried out as markdown headings (`##` a section, `### <n>` . a sub-section) ;
- `warnings` — advisory conditions (e.g. `MISSING_DESCRIPTION` , `DEPRECATED_TEMPLATE`) that never block a render ;
- `attachments` — the aggregated attachments, each qualified by the frame it was declared in.

Failure — a `Diagnostics` whose primary entry is the **earliest blocking diagnostic** : a \$6.2 error code and, where the specification pins one, the pipeline phase that raised it.

How each host surfaces these differs only in idiom — a `Result` in Rust, a thrown `Error` in JavaScript, a raised `ValueError` in Python — but the information is identical. The next chapter shows what to put in each input, value by value, on a complete example ; the three binding chapters after it show the call in each language ; the appendix catalogues the context fields and the full diagnostic list.

Skills and trusted host policy

`analyze` , `tree` and `render` expose `document_type` . The internal composition kind remains `prompt` for compatibility. `@@@template` + `@@@type skill` uses the same inputs and registry ; only its declared nature differs. Python accepts `mixed_inheritance="deny"` ; WASM context accepts `mixed_inheritance: "deny"` ; Rust sets `Context.mixed_inheritance = MixedInheritance::Deny` . Default `allow` reports mixed inheritance/override warnings. Denial is a blocking `MIXED_DOCUMENT_INHERITANCE_FORBIDDEN` diagnostic. A source cannot override the host policy. The core does not write `SKILL.md` or manage local drafts.

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.

Chapter II — The Inputs by Example

The previous chapter named the three inputs; this one shows **exactly what to put in each of them**, value by value. Everything here is drawn from the engine's own sources and its conformance pack — the complete example below is adapted from conformance case L2-CMB-100, so its output is not an illustration: it is what the engine actually produces.

2.1 — A complete call, end to end

Four templates: the entry `a` includes `b`; `b` includes `c`; `c` includes `leaf`. The entry is the `source` argument; the three others go in the `registry`; the `context` binds one parameter set, one toggle, and two parameters in the nested frames.

```
import pool

source = """@@@template a
@@@name A
@@@description combination fixture
@@@section s
@@@include b"""

registry = {
    "b": """@@@template b
@@@name B
@@@description combination fixture
@@@param tone : text = "neutral"
@@@parameter-set f
tone: "formal"
@@@end-parameter-set
@@@section base
b:@@=tone
@@@include c
@@@section? bopt
bopt""",
    "c": """@@@template c
@@@name C
@@@description combination fixture
@@@param cp : text = "cdef"
@@@section s
c:@@=cp
@@@include leaf""",
    "leaf": """@@@template leaf
@@@name Leaf
@@@description combination fixture
@@@param msg : text
@@@section s
```

```

leaf:@@=msg.""",
}

out = pool.render(
    source,
    registry=registry,
    frames={
        "b": {"parameter_set": "f", "section_toggles": {"bopt": True}},
        "b.c": {"parameters": {"cp": "OVR"}},
        "b.c.leaf": {"parameters": {"msg": "DEEP"}},
    },
)
print(out["text"])

```

```

## S

## Base

b:formal

## S

c:OVR

## S

leaf:DEEP.

## Bopt

bopt

```

The headings are the rendered STRUCTURE: # A (A) pairs the entry prompt's `@@@name` with its canonical id, and each retained section carries its own `##` at the point where it renders — including the sections reached through the includes (three of these fixtures happen to name their section `s`). An included prompt gets no title of its own; only the instantiated one does.

Reading it backwards: `tone` came from parameter set `f` selected on frame `b`; `cp` had default `"cdef"` but the frame-keyed value `OVR` won; `msg` has **no** default and was bound on frame `b.c.leaf` (leaving it out raises `UNDEFINED_PARAMETER`); and section `bopt` — declared `@@@section?`, so off by default — was toggled on. The rest of this chapter walks through each argument.

2.2 — `source` — the document to render

The complete text of **one** POO²L document, exactly as it would sit in a `.pool` file: the strict form, `@@@template <id>` first, then the declarations, then the sections. It is a plain string — the engine never reads files, so whatever your application stores (a database row, an editor buffer, an HTTP body) is passed as-is.

The document does not have to be a prompt. The engine reports what it parsed (`validate` returns the kind): a **prompt** renders; an **unstructured** document (no `@@@` constructs) passes through verbatim; a **library** is validated and renders empty; a collection document is not instantiable by itself — you render one of its members.

2.3 — `registry` — what to put in it, and how

The registry is the render's *world*: every other document the render might need. Each entry is one document, exactly like `source` :

- the **value** is the document's complete source text — the same text a `.pool` file would contain, never a fragment;
- the **key** is the document's *address* — what makes it findable.

What must be in it. Every prompt the source `@@@include`s, directly or through other includes; every `@@@inherits` parent, up the whole chain; and, when collections are involved, the `@@@collection` document *plus* one entry per member (see below). A reference to something absent fails with `UNDEFINED_REFERENCE` naming the missing handle.

What may be in it. Anything else. Unreferenced entries cost parsing time but change nothing — so the simple strategy of handing over the *entire* prompt store is correct, and a smaller pre-filtered set is only an optimisation. A document that fails to parse does not poison the render: it is excluded from the world, and its own diagnostic resurfaces only if something actually references it.

The key forms, from the engine's resolution rules :

```
registry = {
  # 1. A plain key — typically the prompt's id.
  "greeting": "@@@template greeting\n@@@name G\n@@@section g\nHi!",

  # 2. A folder-style key: the '/'-separated part is the FOLDER the
  #    document lives in ('marketing'), a scope frame for resolution.
  "marketing/disclaimer": "@@@template disclaimer\n@@@name D\n@@@section d\n...",

  # 3. A collection: the @@@collection document (members are listed by
  #    @@@template lines) under its name...
  "Support": "@@@collection Support\n@@@template welcome\n@@@template
  # ...plus each member as its OWN entry, with the QUALIFIED id.
  "Support.welcome": "@@@template Support.welcome\n@@@name W\n@@@section s\n.
  "Support.closing": "@@@template Support.closing\n@@@name C\n@@@section s\n.

  # 4. A project marker: no document — it flags folder 'teamA' as a
```

```
#    scope boundary the outward folder walk must not cross (§3.22).
"__project__teamA":    "",
}
```

Three engine behaviours worth knowing when you build the map :

- **A prompt is findable under its declared id whatever its key.** The engine indexes each prompt twice : by its full registry key *and* by the id in its `@@@template` line. Key `marketing/disclaimer` with `@@@template disclaimer` is includable as `disclaimer` (folder proximity permitting).
- **Identifiers are case-insensitive.** Keys, ids and include handles are all folded to canonical uppercase (full Unicode: `problème` $\equiv$ `PROBLÈME`), so `@@@include Greeting` finds the `greeting` entry.
- **The key names a header-less document.** If a document has no `@@@template` line, its id is derived from its key (minus the folder prefix) — the key is then not just an address but the identity.

How includes resolve against it — enough to predict what a handle finds: a *bare* handle (`@@@include disclaimer`) is looked up in the current scope frame only, with folder proximity (the referrer's folder, then its ancestors, bounded by the enclosing project); a *qualified* handle (`@@@include Support.welcome`) walks the enclosing scopes outward. One match wins; two candidates at the same level raise `AMBIGUOUS_INCLUDE`; none raises `UNDEFINED_REFERENCE`.

Backing it with a store. Because the registry is a plain map, the natural backend integration is one query away :

```
registry = {p.template_id: p.source for p in template_store.all()}
out = pool.render(source, registry=registry, parameters={...})
```

In JavaScript the same map is a plain object, passed as JSON: `render(source, JSON.stringify({greeting: "@@@template greeting\n..."}), contextJson)` — and `""` (or `"{}"`) when there is nothing to supply. In Rust it is any type implementing `Registry::entries() -> Vec<(String, String)>`; `VecRegistry` wraps an in-memory list, `NullRegistry` supplies nothing.

2.4 — context — example values, field by field

Every field is optional; each line below is a working example value next to its effect. (The full semantics — validation order, precedence, every diagnostic — are catalogued in the reference appendix.)

```
out = pool.render(
    source,
    registry=registry,

    # Values for parameters DECLARED by the root prompt. Scalars are
    # stringified: True -> "true", 3 -> "3". Each value is checked against
```

```

# the declaration (type, select options, min/max/minlen/maxlen/pattern).
parameters={"who": "Claude", "retries": 3, "verbose": True},

# Values for the @@@=NAME system variables – the HOST computes them.
system={"DATE": "2026-07-12", "USER": "jean.dupont", "LANG": "fr"},

# Root-template togglable sections: @@@section? is off unless toggled on,
# @@@section! is on unless toggled off. An off section is fully inert.
section_toggles={"details": True, "legal": False},

# One of the prompt's @@@parameter-set names. Explicit parameters=
# override the set's bindings; declared defaults fill the rest.
parameter_set="production",

# Bindings for declared to-include attachments. The value is OPAQUE
# to the engine (a path, URL, storage key - your convention).
attachments={"spec": "file:./spec.pdf"},

# Depth budget for includes AND parent/inheritance nesting (default 256).
include_depth=32,

# Per-frame inputs for INCLUDED templates, keyed by frame path: the
# dot-joined chain of resolved prompt ids from the root, as in the
# worked example above ("b", "b.c", "b.c.leaf").
frames={"b": {"parameters": {"tone": "warm"}}},

# The folder the SOURCE prompt lives in (" " = root), so its bare
# sibling includes resolve in that folder first.
caller_folder="marketing",
)

```

The same context as the JSON the WebAssembly binding takes — identical field names, except the depth limit sits under `limits` :

```

{
  "parameters":    { "who": "Claude", "retries": 3, "verbose": true },
  "system":        { "DATE": "2026-07-12", "USER": "jean.dupont" },
  "section_toggles": { "details": true, "legal": false },
  "parameter_set": "production",
  "attachments":   { "spec": "file:./spec.pdf" },
  "limits":        { "include_depth": 32 },
  "frames":        { "b": { "parameters": { "tone": "warm" } } },
  "caller_folder": "marketing"
}

```

Two rules of thumb prevent most first-contact errors. Everything you pass must be **declared** by the prompt it targets : an unknown parameter name, an unknown toggle, an unknown attachment, an unknown parameter set and an unreachable frame path each have a dedicated diagnostic (`UNDECLARED_INSTANTIATION_PARAMETER`, `UNK-`

`NOWN_SECTION_TOGGLE`, `UNDECLARED_ATTACHMENT_BINDING`, `UNKNOWN_PARAMETER_SET`, `UNKNOWN_FRAME`) — nothing is silently ignored. And when several sources could bind the same parameter, the order is: call-site include argument, then explicit `parameters`, then the selected parameter set, then the declared default.

2.5 — What comes back

```
{
  "text": "# A (A)\n\n## S\n\n## Base\n\nnb:formal\n\n## S\n\nc:OVR\n\n## S\n\nleaf:DEEP
  "warnings": [],      # e.g. ["DEPRECATED_TEMPLATE: prompt 'b' is deprecated"]
  "attachments": [],   # one {"name","type","kind","frame"} per attachment
                      # declared by a visited frame, e.g.
                      # {"name": "spec", "type": "pdf",
                      #   "kind": "to-include", "frame": "b.c"}
```

`text` is byte-exact and identical across the three bindings; `warnings` are advisory strings (`CODE: detail`); each attachment is qualified by the frame path that declared it ("" = the root prompt). A failed render raises instead — the next chapters show the per-language form of both outcomes.

Partie 2

The Bindings

Chapter III — The Rust Core API

When you embed the engine in a Rust application, you work against `pool-core` directly — the native, canonical API that the WebAssembly and Python bindings (the next two chapters) wrap. All three expose the same shape ; this chapter is the reference form.

Embedding the engine inside a product you distribute, or a service you host commercially, requires the commercial licence — see the Installation Guide, Part 3.

3.1 — The one function

The whole engine is one call :

```
use pool_core::{render, Context, VecRegistry};

let registry = VecRegistry::new(vec![
    ("greeting".into(),
     "###template greeting\n###name G\n###section g\nHello!".into()),
]);

let mut ctx = Context::empty();
ctx.parameters.insert("who".into(), "Claude".into());

match render(source, &ctx, &registry) {
    Ok(out) => {
        println!("{}", out.text);      // the rendered prompt
        // out.warnings      : Vec<Warning>      (struct { code, detail })
        // out.attachments  : Vec<Attachment>    (name, type, kind, frame)
    }
    Err(diags) => eprintln!("{diags}"), // the earliest blocking diagnostic
}
```

`render(source, context, registry)` returns either a `RenderedTemplate` (text + warnings + attachments) or a `Diagnostics` carrying the single blocking error.

3.2 — The three inputs

Source — the `.pool` text to instantiate.

Context — the instantiation inputs for the root frame. Every field is optional (`Context::empty()` renders with declared defaults):

- `parameters: BTreeMap<String, String>` — values bound to **declared** parameters of the root prompt. Values are strings; each is NFC-normalised, then checked against the declaration (type, select options, constraints). Binding an undeclared

name is `UNDECLARED_INSTANTIATION_PARAMETER` ; binding a constant is `IMMUTABILITY_VIOLATION` .

- `system: BTreeMap<String, String>` — values for the system variables (`DATE` , `TIME` , `USER` , `GROUPS` , ...). The host computes them; the engine substitutes them as `text` values, and they never shadow a declared parameter or constant.
- `section_toggles: BTreeMap<String, bool>` — turn toggleable sections (`@@@section?` / `@@@section!`) on or off, overriding their declared default. An off section is inert — even its includes are skipped. A key naming no toggleable section is `UNKNOWN_SECTION_TOGGLE` .
- `parameter_set: Option<String>` — select a named parameter set; explicit `parameters` override the set's bindings, defaults fill the rest.
- `attachments: BTreeMap<String, String>` — bindings for `to-include` attachments; the value (a path, URL, storage key...) is opaque to the engine.
- `include_depth_limit: Option<usize>` — the expansion-depth budget for includes *and* parent /inheritance nesting; `None` uses the engine default (256). Exceeding it is `RESOURCE_LIMIT_EXCEEDED` .
- `frames: BTreeMap<String, FrameInputs>` — per-frame `parameters` , `section_toggles` and `parameter_set` for included frames, keyed by frame path (`"greeting"` , `"greeting.signature"` , ...). A path no include reaches is `UNKNOWN_FRAME` .
- `caller_folder: String` — the folder the entry prompt lives in (`"` = root), so bare sibling includes resolve within that folder.

The reference appendix details each field — stringification, validation order, binding precedence — with examples.

Registry — the entities the source may reference. This is the engine's only contact with the outside world, and it is *injected* : the core never reads files itself.

3.3 — The registry is the integration point

`Registry` is a trait with a single method — «give me the available entity sources as (key, source) pairs.» The core parses and resolves them; the host decides where they come from.

```
pub trait Registry {  
    fn entries(&self) -> Vec<(String, String)>;  
}
```

Two implementations ship with the core :

- `NullRegistry` — supplies nothing (for a standalone prompt with no includes);
- `VecRegistry` — an in-memory list of (key, source) pairs.

For anything else — a database, a content store, an HTTP backend, your existing IziBuddy prompt store — you implement `entries()` over your own source. Because this is the *only* I/O boundary, porting the engine to a new environment never means touching the language logic.

3.4 — What you get back

`RenderedTemplate` has three fields:

- `text` — the assembled prompt, byte-exact;
- `warnings: Vec<Warning>` — advisory conditions, each a struct `{ code, detail }` (e.g. `MISSING_DESCRIPTION`, `DEPRECATED_TEMPLATE`); they never block. A prompt with no `@@@description` yields a `warnings` vector containing `MISSING_DESCRIPTION`, not an empty one;
- `attachments` — the aggregated attachments, each qualified by the frame it was declared in.

A failure is a `Diagnostics` whose primary entry is the **earliest blocking diagnostic**: a \$6.2 error code and, where the specification pins one, the pipeline phase. That is the same contract the CLI and both bindings expose — only the data types change with the host language.

3.5 — The rest of the surface: `validate`, `analyze`, `scope_candidates`

Alongside `render`, `pool-core` exposes three companion functions (all in the crate root):

```
use pool_core::{validate, analyze, scope_candidates};

// Parse gate — no registry, no render.
let (kind, warnings): (DocKind, Vec<Warning>) = validate(source)?;

// Static structure — params, consts, sections, includes; no render.
let a: Analysis = analyze(source)?;
//   a.kind, a.id, a.name, a.description, a.inherits, a.is_virtual, a.is_final,
//   a.params, a.consts, a.sections, a.includes, a.warnings

// Include autocomplete — (handle, kind) pairs reachable from a folder.
let cand: Vec<(String, String)> =
    scope_candidates(source, &registry, "marketing"?;
```

`validate` runs only the parsing phases and returns the document kind plus advisory warnings (or the earliest blocking diagnostic). `analyze` returns the `Analysis` struct — the source's declared structure without resolving includes or rendering. `scope_candidates` builds the world and enumerates the include targets in scope. The bindings project these into their host idiom: the Python binding wraps all four (`validate/analyze` return dicts with an `ok/errors` shape rather than raising), and the WebAssembly binding wraps `validate` and `analyze` as JSON.

Chapter IV — The JavaScript / WebAssembly API

The WebAssembly binding exposes the core to JavaScript — for live preview in a browser, or rendering in Node. It is the same engine that passes the conformance pack, running client-side.

*Building the `pkg/` package (`wasm-pack build ...`) is covered in the *Installation Guide*. This chapter assumes you have it and import from it.*

4.1 — Using it

```
<script type="module">
  import init, { render } from "./pkg/pool_wasm.js";
  await init();

  const source = `@@@template p
@@@name P
@@@param who : text = "world"
@@@section s
Hello @@@=who.`;

  const registry = {}; // { key: sourceString, ... }
  const context = { parameters: { who: "Claude" } };

  try {
    const out = JSON.parse(
      render(source, JSON.stringify(registry), JSON.stringify(context))
    );
    console.log(out.text); // "# P (P)\n\n## S\n\nHello Claude."
    console.log(out.warnings); // ["MISSING_DESCRIPTION: ..."] — the sample has no @@@
    console.log(out.attachments); // [{ name, type, kind, frame }]
  } catch (e) {
    console.error("POOL error:", e.message); // the blocking diagnostic
  }
</script>
```

4.2 — The contract

```
render(source, registryJson, contextJson) -> string
```

- **source** — the `.pool` text.
- **registryJson** — a JSON object `{ key: source }` of the entities the source may include; pass `""` (or `"{}"`) when there are none.

- **contextJson** — the instantiation context, using the same schema as the conformance `context.json`; pass `""` for none.
- **returns** a JSON string `{ text, warnings: string[], attachments: object[] }`.
- **throws** a JavaScript `Error` carrying the blocking diagnostic on failure.

The full context schema — every field optional:

```
{
  "parameters":    { "who": "Claude", "retries": 3, "verbose": true },
  "system":        { "DATE": "2026-07-12", "USER": "jean.dupont" },
  "section_toggles": { "details": true, "legal": false },
  "parameter_set": "production",
  "attachments":   { "spec": "file:./spec.pdf" },
  "limits":        { "include_depth": 32 },
  "frames":        { "greeting": { "parameters": { "lang": "fr" } } },
  "caller_folder": "marketing"
}
```

Parameter and system values may be JSON strings, numbers or booleans — the binding stringifies them the way the engine expects (`3` → `"3"`, `true` → `"true"`); arrays and objects are not scalar values and are dropped. `section_toggles` addresses the root prompt's togglable sections by name; `frames` carries the per-frame `parameters` / `section_toggles` / `parameter_set` for included frames, keyed by frame path (`"greeting"`, `"greeting.signature"`, ...); `caller_folder` is the folder the rendered prompt lives in, so bare sibling includes resolve there. The reference appendix details each field's semantics, validation and diagnostics.

JSON strings in and out keep the boundary stable and language-agnostic: nothing in the contract depends on a particular bundler or framework. A typical live preview re-invokes `render` on each keystroke (it is fast and pure) and shows either `out.text` or the caught error message.

4.3 — `validate` and `analyze`

The WebAssembly binding also exports `validate` and `analyze`. Both take a source and return a **JSON string** — they never throw, reporting grammar problems in an `errors` array instead:

```
import init, { render, validate, analyze } from "./pkg/pool_wasm.js";

const v = JSON.parse(validate(source));
// { ok, kind, warnings: string[], errors: [{ code, phase, message }] }

const a = JSON.parse(analyze(source));
// { ok, kind, id, name, description, inherits, is_virtual, is_final,
//   params: [...], consts: [...], sections: [...], includes: [...],
//   warnings: string[], errors: [...] }
```

`validate` is the parse gate — no registry, no render — powering editor diagnostics ; `analyze` returns the source's static structure (declared params, sections and includes) for the editor's structural views. Because they return data rather than throwing, a `false` `ok` with a populated `errors` array is the normal way a malformed source is reported. (`scope_candidates` is not part of the WebAssembly surface ; it is available in the Rust core and the Python binding.)

Chapter V — The Python API

The Python binding exposes the core as a native CPython module named `pool`. It is the path for server-side integration — and it is how the engine is wired into the IziBuddy backend today: the hand-written resolver has been **removed**, and rendering now goes through this module.

Building and installing the module (`maturin develop` / `maturin build`) is covered in the Installation Guide. This chapter assumes `import pool` works.

5.1 — Using it

```
import pool

src = '''@@@template p
@@@name P
@@@param who : text = "world"
@@@section s
Hello @@@=who.'''

out = pool.render(src, parameters={"who": "Claude"})
print(out["text"])          # "# P (P)\n\n## S\n\nHello Claude."
print(out["warnings"])      # ["MISSING_DESCRIPTION: ..."] — the sample has no @@@descr
print(out["attachments"])   # [{"name", "type", "kind", "frame"}]
```

Includes are resolved from a registry of `{ id: source }`:

```
out = pool.render(
    "@@@template a\n@@@name A\n@@@description d\n@@@section s\n@@@include greeting",
    registry={"greeting": "@@@template greeting\n@@@name G\n@@@section g\nHi!"},
)
```

A blocking diagnostic is raised as a `ValueError` carrying the §6.2 code:

```
try:
    pool.render("@@@template p\n@@@name P\n@@@section s\n@@@=ghost")
except ValueError as e:
    print(e)    # "UNDEFINED_REFERENCE (phase 9): ..."
```

5.2 — The signature

```
pool.render(  
    source: str,  
    registry: dict[str, str] = None,          # { key: source } for includes  
    parameters: dict = None,                  # root-frame parameter values  
    system: dict = None,                      # system variables (DATE, USER, ...)  
    section_toggles: dict[str, bool] = None,  
    parameter_set: str = None,  
    attachments: dict[str, str] = None,       # to-include bindings  
    include_depth: int = None,                # limits.include_depth  
    frames: dict = None,                      # frame-keyed inputs  
    caller_folder: str = None,                # folder of the entry prompt  
) -> dict # {"text": str, "warnings": list[str], "attachments": list[dict]}
```

Argument by argument:

- **registry** — the entities the source may include or inherit from. A key is an addressable path: a prompt id (`greeting`), a folder-style path (`marketing/disclaimer`), or a collection document's key; a prompt is also addressable by its declared id whatever its key.
- **parameters** — values for the root prompt's **declared** parameters. Scalars are stringified as the engine expects: a Python `bool` becomes lowercase `true` / `false`, numbers stringify naturally (`3` → `"3"`), strings pass through. Each value is validated against the declaration — type (`text` / `number` / `boolean` / `date` / `textarea`, plus `select`), select options, constraints (`min` / `max`, `minlen` / `maxlen`, `pattern`).
- **system** — values for the system variables (`DATE`, `TIME`, `USER`, `GROUPS`, ...). The host computes them; they substitute as `text` and never shadow a declared parameter or constant.
- **section_toggles** — `{section_name: bool}` for the root prompt's toggleable sections (`@@@section?` off by default, `@@@section!` on by default), overriding their declared default. An off section is inert — even its includes are skipped.
- **parameter_set** — a named set declared by the prompt. Precedence: explicit `parameters` beat set bindings; declared defaults fill the rest.
- **attachments** — `{name: content_ref}` for `to-include` attachments; the value (path, URL, storage key...) is opaque to the engine. Required attachments left unbound raise `ATTACHMENT_REQUIRED`.
- **include_depth** — the expansion-depth budget for includes *and* parent /inheritance nesting (engine default: 256).
- **frames** — the same inputs for included frames, keyed by frame path: `{"greeting": {"parameters": ..., "section_toggles": ..., "parameter_set": ...}}`, where `greeting.signature` addresses the frame that `greeting`'s own include opens. Call-site arguments override frame-keyed parameters.
- **caller_folder** — the folder the rendered prompt lives in (default: `root`), so bare sibling includes (`@@@include ROLE`) resolve within it.

A fuller call :

```
out = pool.render(
    source,
    registry=store_sources,          # { key: source } from your store
    parameters={"who": "Claude", "retries": 3},
    system={"DATE": "2026-07-12", "USER": "jean.dupont"},
    section_toggles={"details": True},
    parameter_set="production",
    frames={"greeting": {"parameters": {"lang": "fr"}}},
    caller_folder="marketing",
)
```

The return value is a plain `dict`, so it serialises straight to JSON for an API response. The reference appendix details each field's semantics — binding precedence, validation order, and the diagnostics each can raise.

5.3 — `validate`, `analyze`, `scope_candidates`

The Python binding wraps **all four** public functions. Unlike `render`, `validate` and `analyze` do not raise on a grammar error — they report it in an `errors` list, which makes them convenient for editor and migration tooling :

```
v = pool.validate(src)
# {"ok": bool, "kind": str|None, "warnings": [str], "errors": [{"code", "phase", "mess

a = pool.analyze(src)
# {"ok": bool, "kind": str, "id", "name", "description", "inherits",
#  "is_virtual", "is_final",
#  "params": [{"name", "type", "default", "select_options", "constraints",
#              "optional", "required", "label"}],
#  "consts": [{"name", "type", "value"}],
#  "sections": [{"name", "description", "togglable", "default_on", "virtual", "final"}],
#  "includes": [{"handle", "args", "has_args", "section_after"}],
#  "warnings": [str], "errors": [...]}

cands = pool.scope_candidates(src, registry=store_sources, current_folder="marketing"
# [{"handle": str, "kind": str}] - include autocomplete
```

`validate` is a parse gate (no registry, no render); `analyze` returns the static structure (declared params, consts, sections, includes) without resolving includes or rendering; `scope_candidates` lists the `@@@include` targets in scope. IziBuddy uses `analyze` for include analysis and reference resolution, and `render` for the actual output.

5.4 — Wiring it into a backend

Because the registry is injected, the natural integration is to back `registry=` with your existing prompt store: fetch the entities a request needs, hand them to `pool.render(...)` as `{ id: source }`, and return the resulting dict. The engine stays pure; your application owns storage, caching and access control. The rendering itself is deterministic and side-effect-free, which makes it easy to test and to cache.

Partie 3

Diagnostics

Chapter VI — Diagnostics : log and trace

The engine can explain what it did. Two kinds of diagnostics are produced, with opposite audiences and opposite confidentiality :

- **log** — a human-readable, chronological narrative. **Client-readable** (clear).
- **trace** — the structured, detailed record. It is the vendor's IP and is **always sealed** to the vendor's key — it is *never* emitted in clear, and only the vendor can decrypt it.

Both come from one event stream, so they never disagree ; they are two projections of the same thing.

6.1 — The flag string `ltLTaA`

Diagnostics are selected by a short flag string. Lowercase = prompt scope, uppercase = engine + prompt scope (more internal detail). The letter chooses the kind :

Flag	Captures	Output
<code>l</code>	template-level	log (clear)
<code>t</code>	template-level	trace (sealed)
<code>a</code>	template-level	log + trace
<code>L</code>	engine + prompt	log (clear)
<code>T</code>	engine + prompt	trace (sealed)
<code>A</code>	engine + prompt	log + trace

Flags are additive : `lT` = a clear prompt log **and** a sealed engine+prompt trace. An empty string means no diagnostics (the default, zero cost).

*The **scope** (lower/upper) only changes how much detail is captured. The **kind** (log vs trace) decides confidentiality: a log is always clear for the client; a trace is always sealed for the vendor.*

6.2 — Verbosity: `v`, `vv`, `vvv`

The scope flags decide *what* is captured ; the repeatable `v` flag decides *how deep*. Each `v` raises the detail level by one, capped at three :

Level	Flag	Adds
0	(none)	The essentials — instantiation and the final outcome.
1	<code>v</code>	The decisive steps : parameter bindings, sections switched off , include resolutions, inheritance chains, assertion results.
2	<code>vv</code>	The full play-by-play : every section rendered on , the resolution cascade and its candidate count.
3	<code>vvv</code>	Reserved for the deepest internal detail.

`v` combines with any scope. `-lv` is a level-1 prompt log ; `-Tvv` is a level-2 engine+prompt trace (sealed). `-v` on its own (no `l / t / L / T / a / A` scope) produces a clear **stats** summary — a short render report — rather than a prompt log :

```
pool render greeting.pool -lv           # level-1 prompt log
pool render greeting.pool -Lv          # level-2 engine+prompt log
pool render greeting.pool -Tvv         # level-2 engine+prompt trace (sealed)
```

Zero cost when off. Verbosity never slows down a render that does not ask for it. Each diagnostic message is built inside a closure that runs **only** if its scope and level are active — so an inactive level (the default, and every production render) costs a single integer comparison and allocates nothing. A higher level adds messages, never overhead to the levels below it.

6.3 — The Rust API

```
use pool_core::{render_with, Context, NullRegistry};

let result = render_with(source, &Context::empty(), &NullRegistry, "lT");

match &result.outcome {
    Ok(p) => println!("{}", p.text),
    Err(d) => eprintln!("{d}"),
}

// result.debug : DebugOutput
// log:          Option<String>      - clear, client-readable
// trace_sealed: Option<Vec<u8>>     - encrypted to the vendor key (vendor-only)
// trace_note:   Option<String>     - set if a trace was asked for but this
//                                     build cannot seal (the trace is never clear)
// stats:        Option<String>     - clear render summary, produced for `~v`
//                                     with no log/trace scope
```

`render(...)` is exactly `render_with(..., "").outcome` — diagnostics off.

A trace is only produced by a build compiled with the `seal` feature; otherwise `trace_sealed` is `None` and `trace_note` explains why. There is no code path that emits a trace in clear.

6.4 — On the command line

```
pool render greeting.pool -l          # clear log → stderr
pool render greeting.pool -lT        # clear log + sealed trace → greeting.trace.age
```

The sealed `*.trace.age` is an age (<https://age-encryption.org>)-encrypted blob. Only the vendor’s private key opens it — the client can produce it and send it, but never read it. The vendor decrypts with the seal-enabled build :

```
pool unseal --key vendor-private.key greeting.trace.age
```

(Building and installing the `pool` tool, including the `seal` build, is covered in the *Installation Guide*.)

6.5 — In the bindings

The WebAssembly and Python bindings will accept the same flag string as an optional `debug` argument; today, diagnostics are available through the Rust API above and the `pool` CLI. The model is identical everywhere : the log comes back clear, the trace comes back sealed.

6.6 — Why the trace is sealed

The trace records the engine’s internal decisions — the resolution cascade, inheritance linearization, phase-by-phase steps. That is the engine’s implementation IP. Sealing it to the vendor’s public key means a support bundle can carry a complete, replayable diagnostic **without exposing how the engine works**, and without the client (or anyone intercepting it) being able to read it. The encryption is asymmetric : the engine only ever encrypts (the private key never ships), so a leaked public key — or a decompiled binary — changes nothing.

Appendices

Appendix A — Context, Registry and Diagnostics Reference

This appendix is the detailed reference for the three things every host supplies or receives : the **context**, the **registry**, and the **diagnostics**. It mirrors the shapes used by the bindings and documents, field by field, what the engine actually does with each input — binding order, validation, and the diagnostics each field can raise. The canonical definitions are in the specification (§6.1 inputs, §6.2 catalogue).

A.1 — Context (instantiation inputs)

The context carries the instantiation inputs for the **root frame** — the prompt being rendered — plus, under `frames`, the same inputs for any frame an include opens. All fields are optional; an empty context renders the prompt with its declared defaults. In JSON form (as used by the WebAssembly binding and the conformance fixtures) :

```
{
  "parameters":    { "who": "Claude", "count": 3, "verbose": true },
  "system":        { "DATE": "2026-07-12", "USER": "jean.dupont" },
  "section_toggles": { "details": true, "steps.one": false },
  "parameter_set": "production",
  "attachments":   { "spec": "file:./spec.pdf" },
  "limits":        { "include_depth": 32 },
  "frames":        { "greeting": { "parameters": { "lang": "fr" } } },
  "caller_folder": "marketing"
}
```

Field	Purpose
<code>parameters</code>	Values bound to declared parameters of the root prompt.
<code>system</code>	System variables (<code>DATE</code> , <code>TIME</code> , <code>USER</code> , <code>GROUPS</code> , ...).
<code>section_toggles</code>	Turn <code>@@@section?</code> / <code>@@@section!</code> sections on or off.
<code>parameter_set</code>	Select a named parameter set declared by the prompt.
<code>attachments</code>	Bindings for <code>to-include</code> attachments.
<code>limits.include_depth</code>	Maximum include/inheritance nesting depth.
<code>frames</code>	The same inputs, keyed by an included frame's path.
<code>caller_folder</code>	The folder the entry prompt lives in (<code>"</code> = root).

The Python binding takes these as keyword arguments (`parameters=` , `system=` , `include_depth=` , `caller_folder=` , ...); the Rust API takes them as fields of `Context` (where the depth limit is named `include_depth_limit`). The rest of this section details each field.

`parameters` — root-frame parameter values

A flat map of `name → value` . Every key must name a **parameter declared by the root prompt** (`@@@param`); a key that names nothing raises `UNDECLARED_INSTANTIATION_PARAMETER` , and a key that names a constant (`@@@const`) raises `IMMUTABILITY_VIOLATION` — constants are immutable by definition.

Values are strings. The engine's value model is textual ; each binding stringifies host scalars on the way in :

- *Python* — a `bool` becomes lowercase `true` / `false` ; numbers and strings stringify naturally (`3` → `"3"` , `2.5` → `"2.5"`).
- *JavaScript/WASM* — JSON strings, numbers and booleans are accepted and stringified the same way ; arrays and objects are not scalar values and are ignored.
- *Rust* — the map is `BTreeMap<String, String>` ; you pass strings directly.

Every runtime string is normalised to Unicode NFC before use (§4.4.2), then checked against the parameter's declaration, in order :

1. **Type check** (`TYPE_MISMATCH`) against the declared type :

Declared type	Accepted values
<code>text</code> (default)	Any single-line string — no <code>\n</code> or <code>\r</code> .
<code>textarea</code>	Any string, line breaks included (a multi-line <code>text</code>).
<code>number</code>	Parses as a finite float (<code>"3"</code> , <code>"2.5"</code> , <code>"-1e3"</code>).
<code>boolean</code>	Exactly <code>"true"</code> or <code>"false"</code> .
<code>date</code>	<code>YYYY-MM-DD</code> , a real calendar date (leap years included).

1. **Select check** (`SELECT_INVALID_OPTION`) — a `select` parameter's value must be one of its declared options.
2. **Constraint check** (`CONSTRAINT_VIOLATION`) — `min` / `max` for numbers ; `minlen` / `maxlen` (counted in characters, not bytes) and `pattern` (which must match the **whole** value) for everything else.

A declared parameter you do *not* bind falls back to its default ; a parameter that ends up with no value at all only fails **if the rendered text actually references it** — the substitution raises `UNDEFINED_PARAMETER` at that point, not before.

```
{ "parameters": { "who": "Claude", "retries": 3, "verbose": true } }
```

against

```
@@@param who : text
@@@param retries : number min=0 max=10 = 1
@@@param verbose : boolean = false
```

binds `who="Claude"`, `retries="3"` (checked against `[0,10]`), and `verbose="true"`.

system — system variables

A flat map of `NAME → value` supplying the values of system variables — the host computes them, the engine only substitutes. All system values are treated as type `text`. The reserved names are :

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

A prompt may not declare a parameter or constant with one of these names — that is a parse-time `SYSTEM_VARIABLE_OVERRIDE`. Conversely, a system value never shadows anything: in the frame, constants and parameters are bound first, and system entries only fill names that are still free. A substitution `@@@=NAME` that resolves to no parameter, no constant and no supplied system variable raises `UNDEFINED_REFERENCE`.

```
{ "system": { "DATE": "2026-07-12", "USER": "jean.dupont", "LANG": "fr" } }
```

renders `Today is @@@=DATE.` as `Today is 2026-07-12.`

section_toggles — turning sections on and off

A flat map of `section name → bool` for the root prompt's **toggable** sections. A section is togglable when declared with a marker :

Declaration	Toggable	Default
<code>@@@section name</code>	no	always rendered
<code>@@@section? name</code>	yes	off unless toggled on
<code>@@@section! name</code>	yes	on unless toggled off

The key is the section's declared name, including the dotted form for subsections (`steps.one`). The supplied boolean overrides the declared default. An **off** section is inert: its text is not emitted and its includes are **not resolved** — an off section can never raise an include error.

A key that names no togglable section of the root prompt raises `UNKNOWN_SECTION_TOGGLE` (naming a non-toggable section is an error too, not a no-op). Root-level toggles address the root prompt only; sections of an included prompt are toggled through `frames.<path>.section_toggles`.

```
{ "section_toggles": { "details": true, "legal": false } }
```

parameter_set — selecting a named set

The name of a `@@@parameter-set` declared by the root prompt. Its bindings are applied first, then explicit `parameters` override them, then declared defaults fill whatever is still unbound :

parameter-set bindings < explicit parameters < declared defaults (fill only)

Selecting a set the prompt does not declare raises `UNKNOWN_PARAMETER_SET` ; a set that binds a name the prompt does not declare as a parameter raises `UNDECLARED_PARAMETER_SET_BINDING` . Values coming from a set go through the same type/select/constraint validation as explicit ones.

```
{ "parameter_set": "production", "parameters": { "retries": 5 } }
```

selects the `production` set, but `retries` — supplied explicitly — wins over the set's own binding for it.

attachments — binding to-include attachments

A flat map of `attachment name → content reference` . The value is **opaque to the engine** — a file path, a URL, a storage key: the engine never opens it, it only records the binding and reproduces it in the output's `attachments` list. Every key must name an attachment declared by the prompt (`UNDECLARED_ATTACHMENT_BINDING` otherwise), and every *required* `to-include` attachment must be bound (`ATTACHMENT_REQUIRED` otherwise).

```
{ "attachments": { "spec": "file:./spec.pdf" } }
```

limits.include_depth — the nesting budget

The maximum expansion depth. It bounds **all** nesting that opens a frame : include chains, section-targeted self-includes, and `parent /inheritance` hops. When a render would exceed it, the engine raises `RESOURCE_LIMIT_EXCEEDED` instead of overflowing the stack.

Unset, the engine default is **256** — far above anything a real prompt does, low enough to stay safe on constrained stacks (WASM, worker threads). A host may lower it (e.g. a live-preview editor that wants fast failure) or raise it. In Rust the field is `include_depth_limit: Option<usize>` ; in Python the keyword is `include_depth=` ; in the JSON context it sits under `limits.include_depth` .

frames — inputs for included frames

Everything above targets the root frame. When the root includes another prompt, that include opens a new frame — and `frames` lets the host address it directly, keyed by its **frame path**: the dot-joined chain of resolved prompt ids from the root (the root itself is the empty path).

If the root includes `greeting`, and `greeting` includes `signature`, the paths are:

Frame	Path
root	"" (use the top-level fields)
the included <code>greeting</code>	"greeting"
the <code>signature</code> it includes	"greeting.signature"

Each entry carries the per-frame subset of the context:

```
{
  "frames": {
    "greeting": { "parameters": { "lang": "fr" }, "parameter_set": "formal" },
    "greeting.signature": { "section_toggles": { "disclaimer": false } }
  }
}
```

- `parameters` — values for the included prompt's declared parameters. **Call-site arguments win**: a value passed inline at the include (`@@@include greeting(lang=en)`) overrides the frame-keyed one.
- `section_toggles` — toggles for that prompt's togglable sections, same semantics as at the root.
- `parameter_set` — a set declared by *that* prompt.

A `frames` key that no include actually reaches raises `UNKNOWN_FRAME` — a misspelt path fails loudly instead of being silently ignored. Note the path is built from **resolved prompt ids** (e.g. `marketing.welcome` for a collection member), not from the handle as written at the include site.

caller_folder — where the entry prompt lives

The folder path of the prompt being rendered, `""` (the default) for the root. The `source` argument is just text — it carries no location — so a host that renders a prompt stored *inside* a folder passes the folder here; bare sibling includes (`@@@include ROLE`) then resolve within that folder before the outward lookup cascade, exactly as they would for a prompt loaded from the registry under that folder key (§3.21 scope frame).

```
pool.render(source, registry=reg, caller_folder="marketing")
```

makes `@@@include disclaimer` inside `source` find `marketing/disclaimer` first.

Binding precedence, in one place

For each declared parameter of a frame, the engine takes the first value that exists in this order, then validates it :

1. call-site include argument (included frames only) ;
2. explicit value — root `parameters` or `frames.<path>.parameters` ;
3. binding from the selected `parameter_set` ;
4. the parameter's declared default.

Constants are bound before parameters and cannot be overridden by anything ; system variables are bound last and only where the name is still free.

A.2 — Registry

The set of entities the source may `@@@include` or inherit from, supplied as `(key, source)` pairs — a JSON object `{ key: source }` in the WebAssembly binding, a `dict[str, str]` in Python, or a type implementing the `Registry` trait in Rust :

```
pub trait Registry {  
    fn entries(&self) -> Vec<(String, String)>;  
}
```

This is the engine's **only** contact with the outside world, and it is *injected* : the core never reads a file, a database or the network itself. The host enumerates the available sources ; the core parses and resolves them.

Keys are addressable paths. A key can be :

- a plain handle (`greeting`) — typically the prompt's id ;
- a folder-style path (`marketing/disclaimer`) — the `/`-separated part is the folder the entity lives in, a scope frame for the lookup cascade and folder-scoped includes ;
- a collection document's key — its members (declared as separate entries with qualified ids, `Coll.member`) are then addressable by qualified id ;
- the synthetic marker `__project__<folder>` — no document ; it flags that folder as a project boundary (§3.22) the outward folder walk must not cross.

Independently of its key, a prompt is **also indexed by its declared id** (`@@@template greeting` is includable as `greeting` whatever its registry key); a document with no `@@@template` header takes its id **from the key** (minus the folder prefix). Keys, ids and handles are case-insensitive — folded to canonical uppercase, full Unicode (`problème` $\equiv$ `PROBLÈME`). Two entities that both match a handle at the same cascade level raise `AMBIGUOUS_INCLUDE` / `AMBIGUOUS_ENTITY` at resolution — the registry itself accepts duplicates ; ambiguity is judged per lookup. A registry document that fails to parse is **excluded, not fatal** : it cannot poison the render of unrelated templates, and its own diagnostic resurfaces only when something actually references it.

Ship-with implementations in Rust: `NullRegistry` (nothing — a standalone prompt) and `VecRegistry` (an in-memory list). The WASM binding accepts `""` or `"{}"` for an empty registry; the Python binding's `registry=` simply defaults to `none`. Anything else — a database, a content store, an HTTP backend — is one `entries()` implementation away.

```
registry = {
    "greeting":          "@@template greeting\n@@name G\n@@section g\nHi!",
    "marketing/disclaimer": "@@template disclaimer\n@@name D\n@@section d\n...",
}
```

A.3 — RenderedTemplate (the success outcome)

A successful render returns three parts :

- **text** — the assembled prompt, byte-exact. The same inputs always produce the same bytes, in every binding.
- **warnings** — advisory conditions that never block. Each is identified by its §6.2 code; the bindings surface them as `CODE: detail strings` (e.g. `DEPRECATED_TEMPLATE: included prompt 'legacy' is deprecated`).
- **attachments** — the aggregated attachment declarations of every frame the render actually visited, in visit order. Each entry is `{ name, type, kind, frame }`, where `kind` is `fixed` or `to-include` and `frame` is the declaring frame's path (`""` = root, `"greeting"`, `"greeting.signature"`, ... — the same paths the context's `frames` field uses). Frames skipped by an off toggle contribute nothing.

A.4 — Diagnostics

A render either succeeds (text + warnings + attachments) or fails with a blocking **error**. Codes are those of specification §6.2; each failure reports the **earliest** blocking diagnostic, together with the pipeline phase where the specification pins one — the same value the CLI prints (`CODE (phase n): message`) and the bindings surface (a thrown `Error` / `ValueError`).

Warnings (advisory — never block)

Code	Meaning
<code>MISSING_DESCRIPTION</code>	A prompt has no description.
<code>DEPRECATED_TEMPLATE</code>	A deprecated prompt was instantiated, included or inherited.
<code>PARAMETER_NAME_IS_PREFIX</code>	A parameter name is a strict prefix of another.

Errors (blocking)

Parsing & structure: `INVALID_SYNTAX`, `INVALID_BOUNDED_KEYWORD`, `DUPLICATE_DECLARATION`, `SECTION_WITH_SUBSECTIONS_HAS_BODY`, `EXPORT_OF_UNKNOWN_SECTION`, `EXPORT_OF_UNKNOWN_MEMBER`, `INVALID_SECTION_PLACEMENT`, `INVALID_SECTION_ANCHOR`, `EMPTY_SECTION`, `TOGGLEABLE_SECTION_WITHOUT_DESCRIPTION`.

Includes & resolution: `CIRCULAR_INCLUDE`, `AMBIGUOUS_INCLUDE`, `AMBIGUOUS_COLLECTION`, `AMBIGUOUS_ENTITY`, `AMBIGUOUS_USES_TARGET`, `UNDEFINED_USES_TARGET`, `UNDEFINED_REFERENCE`, `UNKNOWN_COLLECTION`, `UNKNOWN_FRAME`, `NOT_EXPORTED`, `INCLUDE_OF_VIRTUAL_SECTION`, `UNENUMERABLE_INCLUDE_TARGET`, `TRANSPARENT_INCLUDE_ON_TEMPLATE`, `RESOURCE_LIMIT_EXCEEDED`.

Parameters, constants & values: `UNDEFINED_PARAMETER`, `UNDECLARED_INCLUDE_PARAMETER`, `UNDECLARED_INSTANTIATION_PARAMETER`, `UNKNOWN_PARAMETER_SET`, `UNDECLARED_PARAMETER_SET_BINDING`, `UNKNOWN_SECTION_TOGGLE`, `IMMUTABILITY_VIOLATION`, `TYPE_MISMATCH`, `CONSTRAINT_VIOLATION`, `INVALID_CONSTRAINT`, `SELECT_INVALID_OPTION`, `SYSTEM_VARIABLE_OVERRIDE`.

Attachments: `ATTACHMENT_NOT_FOUND`, `ATTACHMENT_REQUIRED`, `UNDECLARED_ATTACHMENT_BINDING`.

Assertions & scopes: `ASSERTION_FAILED`, `INVALID_ASSERTION`, `MALFORMED_SCOPE`.

Inheritance & OO: `CIRCULAR_INHERITANCE`, `CIRCULAR_COLLECTION_INHERITANCE`, `CIRCULAR_LIBRARY_DEPENDENCY`, `OVERRIDE_OF_FINAL_SECTION`, `OVERRIDE_OF_FINAL_TEMPLATE`, `OVERRIDE_CONTRACT_VIOLATION`, `SECTION_SHAPE_MISMATCH`, `VIRTUAL_SECTION_HAS_BODY`, `VIRTUAL_TEMPLATE_HAS_BODY`, `UNIMPLEMENTED_VIRTUAL_SECTION`, `UNIMPLEMENTED_VIRTUAL_TEMPLATE`, `VIRTUAL_SIGNATURE_MISMATCH`, `DELETED_SECTION_HAS_BODY`, `DELETE_OF_UNINHERITED_SECTION`, `ALIAS_CONFLICT`, `USES_ALIAS_CONFLICT`.

Resources

The engine lives in the `src/engine/` Cargo workspace :

- `crates/pool-core` — the portable, zero-I/O language core (the Rust API) ;
- `crates/pool-wasm` — the WebAssembly binding (see its `README.md`) ;
- `crates/pool-py` — the Python binding (see its `README.md`).

Companion guides. This book covers the *API* — calling the engine. Getting and installing it (building the `pool` tool, the WebAssembly package and the Python wheel) is the subject of the *POO²L Engine Installation Guide*. The language itself is defined by the POO²L specification, with the *POO²L Language Reference* (book 02) as its reader's companion.

Downloads

- PDF : `08-engine-api-guide_v1.10_en_20261008.pdf`
- EPUB : `08-engine-api-guide_v1.10_en_20261008.epub`
- Markdown : `08-engine-api-guide_v1.10_en_20261008.md`