

POO²L Engine Installation Guide

POO²L Engine Installation Guide

*Getting, Building, Installing and Licensing the
Engine — Common and Commercial Editions*

InZeMobile

TABLE DES MATIÈRES

Partie 1 — The Engine

Chapter I — What the POO ² L Engine Is.....	7
Chapter II — Two Editions: Common and Commercial	10

Partie 2 — Installing the Common Edition

Chapter III — Getting and Building the Engine.....	15
Chapter IV — Running and Verifying the Engine	18
Chapter V — Building the WebAssembly Package	21
Chapter VI — Installing the Python Module	23

Partie 3 — The Commercial Edition

Chapter VII — What the Commercial Edition Adds	27
Chapter VIII — Licensing and Activation.....	29

Appendices

Appendix A — Build Targets and Troubleshooting	33
Resources.....	35

Partie 1

The Engine

Chapter I — What the POO²L Engine Is

The POO²L engine is the reference processor for the POO²L language : it takes a `.pool` source, a set of instantiation inputs, and the entities the source may reference, and produces a fully rendered prompt — or a single, precise diagnostic explaining why it cannot.

It implements the language end to end, across the six maturity levels (L0-L5) of the specification, and is validated against the normative conformance pack: **501 of 511 cases pass, 0 fail** ; the other 10 are documented gaps of the 2026-09-26 regex revision (portable subset, assert-string substitution), marked `PENDING.txt` and run as `xfail` until the engine closes them.

1.1 — What it does

Given three things in, it produces one thing out :

- **a source** — the `.pool` text to instantiate ;
- **instantiation inputs** — parameter values, system variables, section toggles, a selected parameter set, attachment bindings, and limits (the « context ») ;
- **a registry** — the other entities the source can `@@@include` or inherit from.

It returns a **rendered prompt** : the final text, the list of advisory warnings, and the aggregated attachments. On a blocking condition it returns exactly one diagnostic — an error code from the specification's §6.2 catalogue, together with the pipeline phase that raised it.

Everything the language defines is implemented : includes (whole-template, section-targeted, inline and bounded), the resolution cascade, parameters and constants with types and constraints, system variables, parameter sets, declarative assertions, the attachment model, string escapes, prompt inheritance with `parent`, virtual and final sections, togglable sections, collections and libraries, and the full error and warning catalogue.

1.2 — How it is built

The engine is written in Rust and organised around a single principle : **one core, every target**.

At the centre is `pool-core`, a **zero-I/O** library. It never touches a filesystem, a database or the network ; the entities it needs are supplied to it through a small *registry* interface that the host implements. Its only common dependency is a Unicode normalization library (required by the specification for runtime strings) ; a second crate (`age`) is pulled in only when the optional `seal` feature is enabled. That discipline is what lets the *same* core be wrapped, unchanged, for very different environments.

Around the core sit the host shims :

- **pool — the command-line tool**: renders a `.pool` file and runs the conformance pack. A single native executable.
- **the WebAssembly binding**: the core compiled to `wasm32`, callable from JavaScript in a browser or in Node — for live preview.
- **the Python binding**: the core exposed as a native CPython module, for server-side integration.

Because the semantics live entirely in the core, every host renders *identical* output for identical input. There is no second implementation to keep in sync.

1.3 — What this guide covers

This is the **installation guide**. It covers how to *obtain* the engine and get it running :

1. **Installing the common edition** — building and installing the `pool` command-line tool, verifying conformance, and building the WebAssembly package and the Python module. This is the *common edition*: free to use on its own.
2. **The commercial edition** — what it adds, when a licence is required, and how activation works. This matters the moment the engine is embedded inside a product or a hosted service.

What it does **not** cover is how to *call* the engine — the `render` function, the context and registry interfaces, and the per-language API contracts. That is the subject of the companion **API guide**. Here we get the engine onto your machine, into your build, and confirm it works; the API guide shows how to drive it from your code.

The two editions render the same templates. They differ only in the terms under which the engine may be integrated, and in the optional services a commercial arrangement may wrap around it — such as support, guarantees, signed builds, or certification. The next chapter draws that line precisely.

Skill-capable engine deployment

Install bindings built from the same source edition as the host. The 2026-10-08 1.0 draft adds `@@@template + @@@type skill, @@@template-body, @@@end-template` and host policy `mixed_inheritance` (`allow` by default, or `deny`). An older engine cannot be assumed to parse skills even if its package envelope is 1.0. Rebuild Python and WASM together, run the skill conformance fixtures, and distribute the new WASM glue with its matching binary. Keep backups before native storage migrations.

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 — Two Editions : Common and Commercial

The engine ships in two editions. They are the **same processor** — byte-for-byte identical rendering — and differ only in what surrounds it and in the terms under which it may be used.

2.1 — The common edition

The common edition is the engine itself: the core library, the `pool` command-line tool, and the WebAssembly and Python bindings. It is **free to use on its own** — to render templates from the command line, to validate `.pool` sources, to experiment, to learn, and for internal or evaluation use.

It is complete: the six conformance levels (L0-L5), the full diagnostic catalogue, and the same **501 pass / 0 fail / 10 xfail** conformance result as the commercial edition. Nothing in the language is held back.

2.2 — The commercial edition

The commercial edition is the arrangement an organisation would use when it **embeds the engine inside a product or a hosted service**. It does not add language features. Beyond the licence terms, the items below are **optional, forward-looking vendor services** a commercial offering *may* wrap around the engine — they are not capabilities built into the engine itself, and the repository does not implement them:

- a **commercial licence** granting the right to integrate and redistribute;
- **support** with a defined response commitment;
- **indemnification** and contractual warranty;
- **signed release builds** and private security updates;
- a **conformance certificate** for your own customers;
- **continuity terms** — assignment to a transferee, and an open-source fallback if none is found;
- optional hosted components (a managed prompt registry, for example).

The one commercial capability that is actually present in the code is the optional `seal` feature — an encrypted diagnostics trace (see chapter 3 and the *API Guide*). Everything else in the list above describes a possible vendor offering, not shipped functionality.

2.3 — The boundary, in one line

Standalone, internal and evaluation use is free. Integrating the engine into a product you distribute, or a service you host commercially, requires the commercial licence.

The boundary is defined by **use**, not by which artifact you run. Calling the `pool` binary as a subprocess from inside a commercial product is integration just as much as linking the library is; both fall under the commercial licence. Conversely, a developer rendering templates locally needs nothing.

Independently of either edition, the **language itself** already carries a continuity clause — this one is written, not forward-looking. On dissolution without a successor rights-holder, or twenty-four months of silence, the specification relicenses from CC BY-ND 4.0 to CC BY 4.0, so it can be corrected and extended by whoever is left (`standard-governance.md` 7.5). That governs the specification, not the engine's licence terms.

2.4 — At a glance

	Common	Commercial
Language features	All six levels (L0-L5)	All six levels (L0-L5)
Conformance	501 / 511 (10 documented xfail)	501 / 511 (10 documented xfail)
Command-line tool	✓	✓
Library bindings (WASM, Python)	✓	✓
Standalone / internal / evaluation use	✓	✓
Integration into a distributed product	—	✓
Use in a commercial hosted service	—	✓
Support & response commitment	—	optional service
Indemnification & warranty	—	optional service
Signed builds & private security updates	—	optional service
Conformance certificate	—	optional service
Continuity terms (assignment, open-source fallback)	—	optional service

The rows marked *optional service* are forward-looking vendor offerings, not functionality present in the repository today.

The rest of Part 2 covers installing the common edition — building and running the `pool` tool, verifying conformance, and building the WebAssembly and Python packages. Part 3 covers the commercial edition: what it adds, the licensing model, and how activation works in practice.

The exact terms of the commercial licence are a separate legal document. This guide describes how the editions differ and how the engine is installed and licensed; it is not the licence itself.

Partie 2

Installing the Common Edition

Chapter III — Getting and Building the Engine

The common edition is built from source with the standard Rust toolchain. There is no runtime to install and no network access at run time : the result is a self-contained native executable (and, for the bindings, a self-contained library).

3.1 — Prerequisites

Install Rust via `rustup` (recommended — it manages toolchains and cross-compilation targets) :

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
rustup default stable
cargo --version          # any recent stable (1.75+) works
```

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

3.2 — Building the command-line tool

```
cd src/engine
cargo build --release
# → target/release/pool
```

That binary is the deliverable. Copy it anywhere and run it :

```
./target/release/pool render path/to/file.pool
```

To put it on your `PATH` :

```
cargo install --path crates/pool-cli      # installs `pool`
```

3.3 — A single, portable executable

By default the binary links the system C library. For a **fully static** file that runs on any Linux distribution with nothing installed — including a `FROM scratch` container — build for the musl target :

```
sudo apt install musl-tools          # the musl linker, once
rustup target add x86_64-unknown-linux-musl
cargo build --release --target x86_64-unknown-linux-musl
# → target/x86_64-unknown-linux-musl/release/pool  (statically linked)
strip target/x86_64-unknown-linux-musl/release/pool # optional, smaller
```

This works because the whole engine is pure Rust, with no C libraries to link.

Other platforms follow the same pattern with a different target triple :

Platform	Target	Output
Linux (static)	x86_64-unknown-linux-musl	pool
Windows	x86_64-pc-windows-gnu	pool.exe
macOS (Apple silicon)	aarch64-apple-darwin	pool

Cross-compiling to macOS from Linux requires the Apple SDK and is best done on a Mac or in CI; a continuous-integration matrix (Linux / macOS / Windows) is the clean way to produce all three at once.

3.4 — Packaging the artifacts

The repository ships a packaging script that builds a distributable artifact for each target, separately. From `src/engine` :

```
./package.sh cli    [--target <triple>] [--seal]    # native pool binary (stripped)
./package.sh wasm  [--web|--bundler|--nodejs] [--seal]  # wasm-pack package → .tar.gz
./package.sh py    [--seal]                          # maturin wheel
./package.sh all   [--seal]                          # every target whose tool is present
```

Artifacts land in `dist/`. Each target needs its own tool (`cargo` / `wasm-pack` / `maturin`); `all` skips a target whose tool is missing rather than failing.

3.5 — Commercial builds and the vendor key

`--seal` builds with the `seal` feature, which enables the encrypted diagnostics trace (see the *API Guide*, « Diagnostics »). It is **off by default** so the common build stays lean and WASM-friendly.

A seal build embeds the **vendor's public key** to encrypt traces *to* the vendor. The matching private key is **never** part of the engine — it stays on a trusted machine and is used only to open sealed traces. The repository carries a **demo** recipient; before distributing, replace it with the production recipient (the constant in `crates/pool-core/src/seal.rs`) and keep the private key out of the repository (it is `.gitignored` under `keys/`).

3.6 — Verifying your build

The workspace ships the normative conformance pack as an integration test. A plain test run rebuilds the engine and replays the whole pack :

```
cargo test          # the single test bundles the whole conformance pack
```

A green result (test result: ok. 1 passed) means every conformance case passes. For the per-block, per-level breakdown, run the tool against the pack:

```
cargo run --release --bin pool -- conformance
```

P00²L conformance – implemented through Level 5

block	level	pass	fail	skip	xfail	xpass
simple	L0	2	0	0	0	0
	L1	51	0	0	0	0
	L2	43	0	0	0	0
	L3	109	0	0	10	0
	L4	59	0	0	0	0
	L5	27	0	0	0	0
combination	L2	30	0	0	0	0
	L3	75	0	0	0	0
	L4	57	0	0	0	0
	L5	24	0	0	0	0
stress	L2	5	0	0	0	0
	L3	5	0	0	0	0
	L4	2	0	0	0	0
complexity	L2	2	0	0	0	0
	L4	6	0	0	0	0
	L5	4	0	0	0	0

total: 501 pass, 0 fail, 0 skip, 10 xfail, 0 xpass

The report is broken down by **block** (simple, combination, stress, complexity) crossed with **level** (L0-L5). The **xfail** column counts documented spec gaps that are expected to fail; an **xpass** (a pending case that now unexpectedly passes) also fails the run, so that promotions are not forgotten.

3.7 — Offline builds

The core has a single *common* dependency (a Unicode normalization library); the CLI adds none. One further crate (age) is compiled in **only** when the optional **seal** feature is enabled — the default build does not pull it. After the first build fetches the needed crates, everything compiles and runs **offline**. The engine performs no network access at run time.

Chapter IV — Running and Verifying the Engine

The `pool` command-line tool is the simplest way to use the engine. It does two things: render a `.pool` file, and run the conformance pack.

4.1 — Rendering a file

```
pool render path/to/file.pool
```

The rendered prompt is written to **standard output**; any advisory warnings and any blocking diagnostic are written to **standard error**. That separation lets you pipe the prompt cleanly while still seeing warnings:

```
pool render greeting.pool > prompt.txt      # only the prompt is captured
pool render greeting.pool 2> issues.log      # warnings/errors go elsewhere
```

A successful render exits with status `0`. A blocking diagnostic prints a line such as

```
error: UNDEFINED_REFERENCE (phase 9): substitution of an undeclared symbol
```

and exits non-zero, so the tool composes well in scripts and build pipelines.

A worked example

Given `greeting.pool`:

```
@@@template greeting
@@@name Greeting
@@@param who : text = "world"
@@@section s
Hello @@@=who.
```

```
$ pool render greeting.pool
Hello world.
```

The sample declares no `@@@description`, so the engine also emits an advisory warning on **standard error** (it does not change the exit status):

```
warning: MISSING_DESCRIPTION: prompt has no @@@description
```

The default `world` comes from the parameter declaration. To supply a value at the command line you instantiate through a host that accepts a context (the library bindings in the next chapters); the CLI renders with declared defaults and the system context it is given.

4.2 — Running the conformance pack

```
pool conformance                # in-repo pack (default path)
pool conformance /path/to/standard-conformance-tests
```

It prints a table broken down by **block × level** (with `xfail` / `xpass` columns) and a total, and exits non-zero if any case below the implemented level fails — a run also fails on an `xpass` (a pending case that unexpectedly passes), so promotions are not forgotten. This is how you certify a build: a green run is a verifiable, repeatable conformance result.

Note on the default path. With no argument, `conformance` looks for the pack at the location baked in at build time. A binary you have copied elsewhere will not find it there — pass the pack directory explicitly. The `render` command has no such caveat and works anywhere.

4.3 — Exit codes

Code	Meaning
0	Success (render produced ; or conformance fully green)
non-zero	A blocking diagnostic, or a conformance failure / unreadable input

Warnings never change the exit code: a render that emits `MISSING_DESCRIPTION` still succeeds. How warnings are treated — ignored, surfaced, or escalated — is the host's policy, not the engine's.

4.4 — Diagnostics flags

`render` accepts additive single-letter flags to capture diagnostics:

```
pool render foo.pool -l          # human-readable log → stderr
pool render foo.pool -lT         # log + sealed trace → foo.trace.age
```

`l/t/a` (lowercase) are template-scope, `L/T/A` (uppercase) add engine detail; combine them freely (`-lT`). A repeatable `v` raises the detail level (`-lv` , `-lvv` , ...) without any cost when it is absent. The **log** is clear; the **trace** is always encrypted to the vendor's key (it requires a `--seal` build, otherwise a note explains it is unavailable). The full meaning of each flag is in the *API Guide*, « Diagnostics ».

A sealed `*.trace.age` is opaque to whoever produced it. On the vendor side, a seal build opens it:

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

Chapter V — Building the WebAssembly Package

The WebAssembly binding compiles the core to `wasm32` and produces a JavaScript package you can load in a browser or in Node — for live preview. This chapter covers **building and installing** that package; calling its `render` function is covered in the companion API guide.

5.1 — Prerequisites

You need the Rust toolchain (see the previous chapters), the WebAssembly target, and `wasm-pack` (which generates the JavaScript glue):

```
rustup target add wasm32-unknown-unknown    # once
cargo install wasm-pack                      # once
```

5.2 — Building the package

From `src/engine` :

```
wasm-pack build crates/pool-wasm --target web --release
```

This produces `crates/pool-wasm/pkg/`, containing the loader `pool_wasm.js`, the compiled `pool_wasm_bg.wasm`, and TypeScript declarations. Copy that `pkg/` directory into your web project, or publish it to your private registry.

Pick the target that matches how you load modules :

<code>--target</code>	For
<code>web</code>	ES modules, loaded directly in the browser
<code>bundler</code>	webpack / Vite / Rollup
<code>nodejs</code>	Node <code>require</code>

Without `wasm-pack` you can run the two underlying steps by hand — a `cargo build ... --target wasm32-unknown-unknown` followed by `wasm-bindgen` over the produced `.wasm` — but `wasm-pack` is the supported path.

5.3 — Confirming it loads

A minimal smoke test, to confirm the package is built and importable :

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

```
await init();
const out = JSON.parse(render('@@@template p\n@@@name P\n@@@section s\nHi', "", ""));
console.log(out.text); // "# P (P)\n\n## S\n\nHi"
</script>
```

If `out.text` prints, the package is installed correctly. The full calling contract — the shape of the registry and context arguments, the result object, and error handling — is documented in the API guide.

5.4 — Distribution notes

The `.wasm` artifact is self-contained: it embeds the whole engine and needs no server round-trip to render. Ship `pkg/` as a static asset. Because the binding is the *common edition*, building it for internal use or evaluation is free; shipping it inside a product or a commercial service falls under the commercial licence (see Part 3).

Chapter VI — Installing the Python Module

The Python binding exposes the core as a native CPython module named `pool` — the path for server-side integration, including the IziBuddy backend. This chapter covers **building and installing** the module; calling its `render` function is covered in the companion API guide.

It is built as a **stable-ABI (abi3)** extension, so a single wheel works on CPython 3.8 and later.

6.1 — Prerequisites

The Rust toolchain (see the previous chapters) and `maturin` (<https://www.maturin.rs>), the build tool for Rust-based Python extensions :

```
pip install maturin
```

6.2 — Installing into a virtual environment

For local development, `maturin develop` compiles the module and installs it into the currently active virtual environment in one step :

```
cd src/engine/crates/pool-py
maturin develop --release
```

6.3 — Building a distributable wheel

To produce a wheel you can install elsewhere or publish to a private index :

```
cd src/engine/crates/pool-py
maturin build --release
# → target/wheels/poo21-*.whl
pip install target/wheels/poo21-*.whl      # install it wherever you need it
```

Because the extension is `abi3`, the single wheel is valid for every CPython from 3.8 upward — you do not build one wheel per Python version.

6.4 — Confirming it imports

A minimal smoke test :

```
import pool
out = pool.render("@@@template p\n@@@name P\n@@@section s\nHi")
print(out["text"])    # "# P (P)\n\n## S\n\nHi"
```

The `#` and `##` lines are the rendered structure: the prompt's `@@@name` paired with its canonical id, then each section's own title. If that prints, the module is installed correctly. The full calling contract — the keyword arguments for the registry and context, the returned dictionary, and error handling — is documented in the API guide.

6.5 — Distribution notes

The wheel bundles the whole engine; it pulls in no Python dependencies at run time and performs no network access. Building it for internal use or evaluation is free; shipping it inside a product or running it in a commercial hosted service falls under the commercial licence (see Part 3).

Partie 3

The Commercial Edition

Chapter VII — What the Commercial Edition Adds

The commercial edition exists for one situation: **embedding the engine inside a product or a hosted service**. It adds no language features — the rendering is identical — but it provides what an organisation needs to depend on the engine in production, and the right to integrate it.

***Scope of this chapter.** What follows describes a prospective commercial offering. Apart from the licence terms and the optional `seal` feature (the encrypted diagnostics trace, which is the one commercial hook actually present in the code), the services below — support, indemnification, signed builds, the conformance certificate, and any hosted components — are **forward-looking vendor offerings**. They are not implemented in the repository and are not features of the engine itself.*

7.1 — What it adds

The right to integrate. The commercial licence grants the permission the common edition does not: to ship the engine inside a product you distribute, or to run it inside a service you host commercially. This is the substance of the edition; everything below is what makes that dependable.

Support with a commitment. A defined channel and response time, rather than best-effort community help.

Indemnification and warranty. Contractual assurances about the engine — the thing a legal or procurement review asks for before a company will build on a component. This is, in practice, a large part of why organisations take the licence: it converts an unmanaged risk into a contracted one.

Signed release builds and private security updates. Binaries and wheels signed so your own build pipeline can verify provenance, plus advance notice and fixes for security-relevant issues.

A conformance certificate. A document attesting the engine's conformance result, which you can present to your own customers and auditors — without exposing any source.

Optional hosted components. Where useful, managed services that the engine alone does not provide could be offered: a hosted prompt registry, a rendering service, a certification endpoint. These would be genuinely separate from the offline core and are the natural place for value that depends on the vendor's infrastructure. None of them ships in the repository today.

7.2 — What it does *not* change

The commercial edition is the **same processor**. Same six levels (L0-L5), same 501 / 511 conformance (the other 10 are documented gaps, xfail), same diagnostics, same output for the same input. You are not buying a more capable engine ; you are buying the right to embed it, plus the support and guarantees that make embedding safe. A prompt that renders one way under the common edition renders identically under the commercial one — by design, since both are the one core.

7.3 — Why the model is shaped this way

The engine is a pure, offline library: its sources, where published, can be read and recompiled, so there is no purely *technical* lock that would make integration impossible without paying. That is deliberate, and it is why the commercial edition is built around **value an integrator cannot reproduce alone** — a licence with legal weight, support, indemnification, signed provenance, certification, continuity terms, and hosted services — rather than around a removable runtime check.

Continuity belongs in that list for the same reason indemnification does : it is the answer to a procurement question no amount of engineering can settle. What happens to the licence if the vendor stops? An integrator cannot arrange that alone — only the rights-holder can commit to it in advance, by undertaking to assign the asset and, failing a transferee, by letting the sources fall to an open-source licence. The reason that second half works at all is that it is a grant rather than a delivery : it takes effect by the contract itself, in the branch where there is nobody left to perform anything. The specification's own continuity is already written down (`standard-governance.md` 7.5); the engine's belongs to the commercial licence document, alongside the other terms of this chapter.

In other words : the common edition is genuinely free for standalone use, and the commercial edition sells the things a business actually needs to ship and stand behind a product built on the engine. The next chapter covers when the licence is required and how activation works.

Chapter VIII — Licensing and Activation

This chapter covers two things: **when** a commercial licence is required, and **how** a licence is activated and shown.

This is an operational description, not the licence text. The binding terms live in the commercial licence agreement, which governs in case of any difference.

Status. *The two-credential model described below is a **forward-looking** design, not an implemented mechanism: the engine ships **no activation code**, and the only licensing code in this repository belongs to the IziBuddy application (a separate product), not to the engine. Treat this chapter as the intended shape of a future commercial arrangement rather than a description of current behaviour.*

8.1 — When the licence is required

The dividing line is **use**, not artifact :

- **Free** — standalone use of the `pool` tool ; internal use ; evaluation ; learning ; rendering templates for yourself or your team.
- **Licensed** — integrating the engine (CLI, library, or any binding) into a product you distribute, or running it inside a service you host commercially.

Running the `pool` binary as a subprocess from a commercial product is integration just as linking the library is. The question is never *which file you run*, but *whether the engine is part of something you ship or sell*.

If you are unsure which side of the line you are on, that is exactly the conversation the commercial licence is for.

8.2 — How activation works : two credentials

A licence is represented by **two distinct credentials**, with opposite properties. Keeping them separate is the whole point — one is safe to show, the other must never be shown.

1. The licence certificate — public, displayable

A signed document stating who is licensed and under what terms (organisation, edition, validity, seats/scope). It is :

- **non-secret** — showing it grants no power, so it is safe to display in an About box, an admin page, or a footer ;
- **self-verifying** — it is signed with the vendor's private key, and the product verifies it offline with the embedded public key. No call home is required to confirm « this licence is genuine and valid until ... » ;

- **bound to the licensee** — the certificate names the licensed organisation, so copying it into someone else’s product proves nothing for *them* ; the signature only attests authenticity for the named party.

The certificate **attests**. It never **authorises** an action — it cannot unlock a server or a feature, because it carries no secret.

2. The API secret — private, never displayed

A separate secret credential used to *authenticate* to the vendor’s services (private updates, the support portal, any hosted component). It :

- **must stay secret** — anyone who sees it could impersonate the licensee, so it lives in server-side configuration, never in the UI, never in logs, and is revocable ;
- **carries the power** — it, and only it, opens a door on the vendor side.

The rule that keeps them apart

*The **displayed** credential only ever states that things are in order. It never authorises anything. Only the **secret** credential authenticates and unlocks. Never make any action depend on the credential you display.*

This is the standard public-identifier / secret split (the same shape as a public client id versus a client secret). It lets a licensee prove, visibly, that their deployment is in order, while the credential that actually grants access stays hidden — and lets the vendor look a customer up from the public certificate without that certificate being usable by anyone else.

8.3 — A note on enforcement

On an offline, source-available engine, an embedded check can be removed by a determined party. The activation mechanism above is therefore not an anti-piracy lock ; it is **entitlement management** — it ties a paying customer to their account and to the support, updates and guarantees they are owed.

Compliance, for serious organisations, is driven by something stronger than a runtime check: the legal weight of the licence, the indemnification they want, and the liability that unlicensed dependencies create in audits, funding and acquisitions. The licence makes integration a decision a business makes deliberately — which is exactly what it is meant to do.

Appendices

Appendix A — Build Targets and Troubleshooting

A quick reference for cross-compilation targets and the build problems most likely to come up. (The engine's *calling* interface — context, registry, diagnostics — is documented in the companion API guide.)

A.1 — Build targets

The `pool` binary and the bindings are produced with the standard Rust target triples :

Goal	Target triple	Output
Linux (dynamic)	<code>x86_64-unknown-linux-gnu</code> (default)	<code>pool</code>
Linux (fully static)	<code>x86_64-unknown-linux-musl</code>	<code>pool</code>
Windows	<code>x86_64-pc-windows-gnu</code>	<code>pool.exe</code>
macOS (Apple silicon)	<code>aarch64-apple-darwin</code>	<code>pool</code>
macOS (Intel)	<code>x86_64-apple-darwin</code>	<code>pool</code>
WebAssembly	<code>wasm32-unknown-unknown</code>	<code>.wasm</code> (+ JS via <code>wasm-pack</code>)

Add a target once with `rustup target add <triple>`, then build with `cargo build --release --target <triple>`. Cross-compiling to macOS from another OS needs the Apple SDK and is best done on a Mac or in CI.

A.2 — Common build issues

cannot find linker / musl build fails. The static Linux build needs the musl linker: `sudo apt install musl-tools` (Debian/Ubuntu), then `rustup target add x86_64-unknown-linux-musl`.

Windows cross-build fails to link. Install the MinGW toolchain: `sudo apt install mingw-w64`, then `rustup target add x86_64-pc-windows-gnu`.

wasm32 target missing. `rustup target add wasm32-unknown-unknown`; the JS glue additionally needs `cargo install wasm-pack`.

maturin cannot find a Python interpreter. Activate a virtual environment first, or point maturin at an interpreter. The `abi3` build does not link a specific `libpython`, so a CPython ≥ 3.8 in scope is enough.

First build seems to hang on « Updating crates.io index ». The core has one *common* dependency (a Unicode normalization library); a second crate (`age`) is fetched **only** when the optional `seal` feature is enabled. They are fetched once. After that, builds work offline.

A.3 — Verifying and the conformance default path

`cargo test` (from `src/engine`) replays the whole conformance pack as a single test; a green result is your verification. `pool conformance` prints the breakdown by block × level (with `xfail / xpass` columns) — currently 501 pass, 0 fail, 10 xfail (511 cases; the ten xfail are documented gaps marked `PENDING.txt`).

One caveat for a *moved* binary: with no argument, `pool conformance` looks for the pack at the path baked in at build time, which will not exist if you copied the executable elsewhere. Pass the pack directory explicitly (`pool conformance /path/to/standard-conformance-tests`). The `pool render` command has no such caveat and works anywhere.

A.4 — Offline operation

After the one-time dependency fetch, every build and every run is offline: the engine performs no network access, reads no configuration at run time, and needs no services. A built `pool` binary, a `pkg/` WebAssembly package, or an installed wheel each contains the whole engine.

Resources

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

- `crates/pool-core` — the portable, zero-I/O language core ;
- `crates/pool-cli` — the `pool` command-line tool and conformance runner ;
- `crates/pool-wasm` — the WebAssembly binding (see its `README.md`) ;
- `crates/pool-py` — the Python binding (see its `README.md`) .

The language is defined by the POO²L specification and its conformance pack ; the *POO²L Language Reference* (book 02) is a reader's companion to it.

Companion guides. This book covers *installation*. Calling the engine from your code — the `render` function, the context and registry interfaces, and the per-language API contracts — is the subject of the *POO²L Engine API Guide*.

Verifying conformance: from `src/engine`, run `cargo test` (asserts the whole pack) or `cargo run --release --bin pool -- conformance` (per-level report).

Downloads

- PDF : `07-engine-installation-guide_v1.7_en_20261008.pdf`
- EPUB : `07-engine-installation-guide_v1.7_en_20261008.epub`
- Markdown : `07-engine-installation-guide_v1.7_en_20261008.md`