Zen Working Group
Request for Discussion and Implementation Record

ZDS 0014
Implementation Specification

The zenfmt Python Library: API and Implementation

STATE
committed

CREATED
2026-08-08

LAST UPDATED
2026-08-08

INTENDED STATUS
Committed

AUTHORS
Vikrant Rathore
Ronak Rathore (assistance)

DISCUSSION

The implemented Python API, native boundary, uv workflow, packaging, benchmark, and release contract

LABELS
python, api, packaging, security

Status of This Memo

This document is an internal Zen discussion record authored in Typst and tracked in git. It intentionally follows RFC-style structure so the design scope, rationale, trade-offs, and operational constraints remain explicit. Documents that still use the placeholder number XXXXX are provisional drafts. Numbered ZDS documents are part of the project record.

Contents

124. Abstract

zenfmt has a carefully bounded Zig conversion engine and command-line tool, but Python applications cannot yet use that engine through a native, structured API. Calling the CLI through `subprocess` discards type information, requires temporary-file conventions for in-memory data, makes error handling depend on process behavior, and cannot provide a Requests-quality developer experience.

This record specifies the `zenfmt` Python distribution and import package: a small, fully typed Python layer over a versioned C ABI implemented by a Zig bridge and loaded with Python's standard-library `ctypes` module. The common operation is one `zenfmt.convert(...)` call. It accepts paths, bytes-like objects, and binary readers; returns an immutable conversion result containing artifact bytes, resources, the canonical manifest, and structured reports; and raises a compact exception hierarchy on failure. A reusable immutable `Converter` carries policy such as resource limits and strictness without introducing global configuration.

The Python project lives in this monorepo, uses `uv` for Python environment, dependency, lock, build, and publish workflows, Hatchling as the PEP 517 backend needed for platform wheels, Ruff for linting and formatting, and pytest for tests. The root `zig build` graph remains the repository's orchestration layer and owns native compilation. Releases publish prebuilt wheels and a buildable source distribution to PyPI under the same version as the Zig release; the first public release is version `0.1.0`, shared by the engine, CLI, native bridge, and Python distribution.

This record is the normative implementation blueprint for that library. Its scope includes the engine changes, native bridge, Python package, root build integration, tests, documentation, platform artifacts, and release workflow needed to deliver it. The record entered discussion before any of those implementations existed; that staging fact does not place implementation outside the record's scope. Once accepted, the delivery phases and acceptance gates below define the work required to move it to `committed`.

125. Introduction

Python is a natural embedding environment for document conversion: web services receive uploads, data pipelines normalize corpora, notebooks inspect results, and test suites need deterministic fixture conversion. Those callers should not have to understand the CLI's argument grammar, parse terminal text, or discover the adjacent-manifest convention before completing their first conversion.

The target quality bar is the best part of libraries such as Requests, Flask, and Stripe's Python library:

- the shortest call expresses the common operation;
- behavior is unsurprising to a Python developer;
- advanced policy is discoverable through named, typed objects;
- errors preserve stable machine data and also read well in a traceback;
- defaults are safe, deterministic, and free of ambient configuration;
- import-time behavior is quiet and cheap;
- documentation and type information ship with the package.

This surface does not replace the Zig API. It adapts the default zenfmt bundle for Python while preserving the engine's format detection, lowering, strictness, manifest, report, resource-limit, and atomic-publication contracts. ZDS 0002 remains authoritative for the engine and ZDS 0013 remains authoritative for the layered IR and writer lowering. When this record and those records disagree about conversion semantics, the engine records win; this record decides how Python represents them.

The word **universal** describes the API shape, not a false claim that every possible document format is already implemented. The Python library exposes every reader and writer in the bundled zenfmt release and discovers that set from native capability metadata. Adding a format to the default Zig bundle makes it available through the same Python call without adding another format-specific Python function.

126. Terminology and Scope

- **distribution**: the project installed from PyPI, named `zenfmt`;
- **import package**: the Python package imported with `import zenfmt`;
- **Python layer**: public Python objects plus private validation, marshalling, model, exception, and native-loading modules;
- **native bridge**: a small Zig shared library exposing a versioned C ABI over the default zenfmt bundle;
- **engine**: the Zig conversion engine defined by ZDS 0002 and ZDS 0013;
- **memory conversion**: a conversion whose artifact and embedded resources are returned as Python-owned bytes;
- **path conversion**: a conversion published atomically to an explicit output path with its manifest and embedded resources;
- **report**: zenfmt's structured diagnostic value, whether note, warning, or error;
- **manifest**: canonical JSON carrying provenance, reports, document metadata, facets, media, and plugin preservation data;
- **capability metadata**: native, machine-readable descriptions of the readers, writers, extensions, and defaults compiled into this release;
- **source distribution**: the PyPI archive containing both the Python source and the Zig sources needed to build the native bridge;
- **wheel**: a platform-tagged Python binary distribution containing the Python layer and one native bridge built for that platform.

In scope:

- implementing the complete first release described by this record;
- the engine additions required for bounded, complete in-memory publication;
- the versioned native bridge and its ABI contract;
- the installable `zenfmt` Python package and public API;
- the first public Python conversion API and its compatibility rules;
- input, output, result, report, manifest, resource, limit, and error models;
- the Python-to-Zig boundary and ownership model;
- Python project layout and root `zig build` integration;
- uv, Hatchling, Ruff, and pytest workflows;
- automated native, Python, integration, wheel, and source-distribution tests;
- platform-wheel, source-distribution, and PyPI release requirements;
- user documentation and public API reference material;
- the protected PyPI publication workflow and optional TestPyPI rehearsal;
- security, concurrency, testing, and support policy.

Out of scope:

- exposing the document IR for Python mutation;
- Python-authored readers, writers, or filters;
- calling Python from Zig;
- a hosted conversion service or network client;
- asynchronous conversion, cancellation, or progress callbacks;
- process sandboxing;
- changing any source-format mapping or writer lowering decision.

126.1. Normative implementation contract

Requirements expressed with **must** or **required**, together with declarative contract statements such as "the loader verifies," are binding for the first release. **May** identifies a permitted choice, and **future** identifies work that requires another record or amendment. An implementation is not complete when the public function merely works on a developer machine; it is complete only when every deliverable and acceptance gate in this record is satisfied.

Deliverable	Required implementation outcome
`core/`	Bounded memory publication with embedded resources, the `max_output_bytes` limit enforced in every output mode, bridge-facing report serialization carrying `exit_class`, and the writer text/binary emission declaration.
`bindings/python/`	A versioned, fuzzed Zig C ABI over the default bundle, with opaque result ownership and no Python callbacks.
`python/src/zenfmt/`	The typed public API, immutable models, validation, exceptions, capability discovery, and secure private loader specified here.
build graph and Python subproject	The self-contained uv project under `python/` (`pyproject.toml`, `uv.lock`, the Hatchling adapter), named `zig build python-*` orchestration steps at the root, and the canonical `build.zig.zon` version plumbed into the CLI, bridge, and Python metadata.
tests	Native ABI tests, pytest unit/integration tests, parity tests, adversarial tests, and installed wheel/sdist tests.
benchmarks	The existing corpus benchmark extended with the clean-installed wheel, plus cold-start, warm in-process, memory/path, and concurrent Python API measurements.
documentation	A quickstart, security guidance, a generated public API reference whose examples are exercised by tests, and the updated reference and benchmark chapters of the book.
release	The new wheel and release workflows, the required platform wheels, standalone sdist, standalone CLI archives, version parity, attestations, protected PyPI publication, and one checksummed GitHub release.

127. Problem Statement

The CLI is a good human and shell interface, but it is not a Python library contract. A wrapper built directly around `subprocess` would inherit several problems:

- failures arrive as exit codes, text, and partial operating-system state instead of typed Python exceptions with structured reports;
- byte input and byte output require pipes or temporary files, while container formats and media resources need seekable or multi-file behavior;
- users must reproduce format selection, output naming, strictness, and manifest conventions in application code;

- every call pays process startup and loses safe in-process concurrency;
- executable lookup through `PATH` can select an unintended version;
- the CLI and wrapper can drift because their option grammars become a second informal ABI.

Exposing the Zig implementation directly through CPython's extension API would solve process startup but create a different coupling: every native function would also own Python reference-count, object-layout, exception, and stable-ABI concerns. The conversion engine does not need those concerns. Its natural portable boundary is a small C ABI using byte slices, native paths, canonical JSON, and opaque result ownership.

The packaging problem is equally important. A Python package containing native code cannot honestly ship as `py3-none-any`; users need a wheel whose platform tag matches the bundled library or a source distribution that can actually rebuild it. A successful design must keep those artifacts synchronized with the Zig release and make a clean-wheel installation behave the same as a checkout.

128. Goals and Non-Goals

128.1. Goals

- Make `zenfmt.convert(source)` the complete common path: safe defaults, automatic input detection, the default writer, structured diagnostics, and no filesystem output unless the caller supplies an output path.
- Present idiomatic Python values: `pathlib.Path`, bytes, immutable slotted data classes, string enums, context managers only where resources require them, and ordinary Python exception chaining.
- Return the whole artifact ensemble. An in-memory conversion must not drop embedded media merely because it has no destination directory.
- Preserve canonical native reports and manifests without forcing callers to parse terminal text.
- Apply the Elm-style diagnostic contract from ZDS 0002 to every Python-facing failure. Each message states what happened, where it happened, what was or was not produced, and at least one concrete direction rendered under `What you can do:`, matching the engine's text renderer.
- Keep conversion semantics identical across Python, the Zig library, and the CLI for the same source, formats, limits, strictness, and release.
- Keep the runtime package dependency-free beyond the Python standard library and its bundled native bridge.
- Release the GIL while native conversion runs and make independent calls safe from multiple Python threads.
- Load only the native library shipped with this distribution and verify its ABI and release version before use.
- Preserve zenfmt's bounded-resource, no-network, no-reference-following, and atomic path-output guarantees.
- Add a native `max_output_bytes` limit, defaulting to 512 MiB, checked while a writer emits in every output mode. The current input, decoded-text, lowering-work, and resource limits do not bound artifact bytes themselves.
- Provide precise annotations, a `py.typed` marker, stable public exports, and copy-pasteable documentation.
- Use one monorepo version and one root Zig orchestration graph.
- Use uv for the Python environment, lockfile, commands, build frontend, and publication; Ruff for both linting and formatting; and pytest for tests.

- Publish artifacts that are reproducible enough to compare by digest and traceable to a tagged source revision.
- Extend the repository benchmark so the shipped zenfmt wheel is measured as a first-class implementation alongside the native CLI, with separate cold and warm Python API results and verified output parity.

128.2. Non-Goals

- Recreating zenfmt's AST as a graph of Python objects.
- Matching Pandoc's Python ecosystem or providing Python filters in the first release.
- Treating a Python string as document contents. A string is a path; text content is encoded explicitly by the caller.
- Hiding warnings. Successful conversions return every native report in order.
- Using Python's `warnings` module for conversion reports; global warning filters and text-only messages would discard zenfmt data.
- Guaranteeing rollback after writing to an arbitrary user-provided stream. The first release therefore returns bytes or publishes to a path and does not accept output streams.
- Loading system plugins, searching `PATH`, reading a configuration file, or consulting environment variables at runtime.
- Making `pip install` compile Zig on platforms for which an official wheel is available.
- Promising a platform before its wheel passes the release gates in this record.

129. Implementation Architecture

The library has three deliberately narrow layers:

1. The public Python layer validates Python arguments, turns successful native output into immutable models, and turns failed native output into typed exceptions. It contains no document-format logic.
2. A private `ctypes` adapter loads the packaged shared library by absolute path, validates its ABI, marshals one conversion request, copies result bytes into Python ownership, and frees the native result.
3. The native bridge translates the versioned C ABI into the existing Zig `ConvertOptions` and `Conversion` contracts, plus one memory-artifact sink that returns embedded resources instead of projecting them onto disk.

The default call is a memory conversion:

```
import zenfmt

conversion = zenfmt.convert("report.docx")
print(conversion.text)
for report in conversion.reports:
    print(report.code, report.problem)
```

Supplying an output path selects transactional path publication:

```
conversion = zenfmt.convert(
    "report.docx",
    output="build/report.md",
    strict="structure",
)

assert conversion.path.name == "report.md"
assert conversion.manifest.artifact.format == "markdown"
```

Byte input is explicit and never causes an implicit filesystem probe:

```
conversion = zenfmt.convert(
    uploaded_bytes,
    name="upload.docx",
    to="markdown",
)
markdown = conversion.content
```

The examples in this record are normative API acceptance examples. The implementation must make them execute unchanged with the behavior described here.

129.1. End-to-end invariants

For every call, the layers preserve these invariants:

- The Python layer never invents a reader, writer, report, loss, manifest field, or resource digest.
- Invalid Python arguments fail before native work begins and raise `TypeError` or `ValueError`.
- Expected conversion failure produces a `ConversionError` carrying all native reports and no successful `Conversion` object.
- Memory failure returns no artifact bytes or resources; native partial stream state is an internal defect because the bridge stages memory output.
- Path failure publishes no manifest. Publication follows the engine's staged artifact, media, then manifest protocol.
- Success returns exactly one canonical manifest. Path success also commits those same manifest bytes beside the artifact.
- Python copies every borrowed native byte slice before freeing the result.
- A result owns only Python values and remains valid after the native handle is released.

130. Public Python API

130.1. Package identity and imports

The PyPI distribution and top-level import package are both named `zenfmt`. The supported public names are enumerated in `zenfmt.__all__`; modules and names beginning with an underscore are private. The package ships `py.typed`, so its inline annotations are available to type checkers without a separate stubs distribution.

Importing `zenfmt` defines Python objects and reads installed distribution metadata, but does not load the native library. The first capability query or conversion loads and verifies the bridge. Lazy loading keeps imports cheap and lets documentation tools inspect the package, while an unusable wheel still fails at the first operation with a focused `NativeLibraryError`.

The package exposes:

Name	Contract
`convert`	One-shot conversion using default policy.
`formats`	Deterministic tuple of native reader and writer capabilities.
`Converter`	Immutable, reusable conversion policy and capability view.
`Conversion`	Successful artifact, resources, manifest, and reports.
`Format`	One compiled reader/writer capability.
`Limits`	Immutable mirror of the engine's public resource limits.

Name	Contract
`Strictness`	The `off`, `content`, `structure`, and `exact` grades.
`Report`, `Direction`, `Context`	Immutable structured diagnostics.
`Manifest`	Typed access plus lossless canonical JSON.
`Resource`	Embedded or external artifact resource.
`ZenfmtError` and subclasses	The stable library exception hierarchy.
`__version__`	The installed PEP 440 distribution version.

130.2. The conversion call

The conceptual signature is:

```
zenfmt.convert(
    source,
    *,
    to=None,
    from_=None,
    output=None,
    name=None,
    strict=False,
    limits=None,
    overwrite=False,
    preserve_facets=False,
) -> Conversion
```

Every parameter after `source` is keyword-only. This protects callers when the API gains optional policy and makes security-sensitive values visible at the call site.

Parameter	Meaning
`source`	A path, bytes-like object, or binary reader as defined below.
`to`	A canonical writer id, extension-like alias, or `Format`; `None` selects the bundle's declared default writer when no output suffix decides.
`from_`	A canonical reader id, extension-like alias, or `Format`; the trailing underscore is required because `from` is a Python keyword.
`output`	`None` for a memory result, or a path-like object for transactional publication.
`name`	A safe display name and extension hint for bytes or a binary reader. It is rejected for path input.
`strict`	`False`, `True`, a `Strictness`, or its string value. `True` means `content`, matching bare `--strict`.
`limits`	A `Limits` value. `None` uses the engine defaults.
`overwrite`	Whether an existing artifact ensemble may be replaced; `False` by default.
`preserve_facets`	Whether the manifest carries full facet rows instead of the default digest-and-count summaries.

Strings naming formats are ASCII case-insensitive. A single leading dot is ignored, so `"DOCX"` and `".docx"` resolve through the same capability table as `"docx"`. Results always expose the canonical

native format id. Unknown or ambiguous aliases become structured `UnknownFormatError` failures with the engine's suggestions; the wrapper does not maintain a second format table.

130.3. Source semantics

Python value	Behavior	Manifest and detection name
`str` or `os.PathLike`	Passed to the native path conversion without pre-reading the file.	The path basename. An adjacent digest-bound manifest may be loaded by the engine.
`bytes`, `bytearray`, `memoryview`	Passed as in-memory bytes. Mutable or non-contiguous buffers are copied before the GIL is released.	`name` if supplied, otherwise `"<memory>"`. No adjacent file is inspected.
binary reader	Consumed from its current position in bounded chunks until EOF. The wrapper neither seeks nor closes it. `read()` must return bytes.	`name` if supplied, otherwise a sanitized basename from its `.name` attribute when that attribute is already a string, otherwise `"<stream>"`. The attribute is never opened or resolved.

A Python `str` is always a filesystem path, never inline text. This removes the most dangerous ambiguity in a converter API. Callers convert text explicitly:

```
conversion = zenfmt.convert(
    "A heading\n=========\n".encode(),
    from_="rst",
    to="markdown",
)
```

Binary readers are capped while reading at `max_input_bytes`; the wrapper reads at most one byte past the cap to distinguish exact-boundary EOF from overflow. On overflow it stops consuming the reader and submits those bounded bytes to the bridge so the native engine produces the canonical limit report; Python does not invent a parallel report. The wrapper never calls `fileno`, reopens `.name`, follows a URL, or spills to a predictable path.

An explicit `name` must be a non-empty display basename without a directory separator, NUL, or control character. This prevents an in-memory caller from placing an absolute local path into diagnostics or the reproducible manifest.

On POSIX, path-like values returning `bytes` remain raw operating-system path bytes. On Windows, paths must be `str`/Unicode as required by normal Python filesystem practice. The private ABI has explicit native-path encodings so a filename is never round-tripped through a lossy locale encoding. Embedded NUL characters are rejected before native entry.

130.4. Destination semantics

`output=None` is deliberately side-effect-free with respect to output files. The conversion stages the artifact and all embedded resources in native-owned memory, computes their digests and canonical manifest, then copies the complete ensemble into the result. The Python layer derives the artifact name from the source display name and the writer's primary extension taken from capability metadata — `artifact.<extension>` when no usable stem exists — and passes that

basename across the ABI, where the engine uses it for deterministic resource naming and the manifest.

An output path delegates to the engine's transactional publication. Success means the artifact, its embedded media tree, and `<output>.zenfmt.json` were published as one manifest-vouched ensemble. Existing artifact, manifest, or media targets cause `DestinationExistsError` unless `overwrite=True`.

Arbitrary output streams are not accepted in the first release. They cannot promise rollback after a Python writer accepts a prefix, and buffering an unbounded result in Python merely recreates memory conversion with a less clear contract. A caller that owns this trade-off can write `conversion.content` after successful memory conversion.

130.5. Reusable policy with `Converter`

`Converter` is an immutable, slotted, thread-safe value. It validates policy at construction and reuses the lazily loaded native bridge:

```
converter = zenfmt.Converter(
    strict=zenfmt.Strictness.STRUCTURE,
    limits=zenfmt.Limits(max_input_bytes=64 * 1024 * 1024),
)

first = converter.convert("one.docx")
second = converter.convert("two.pdf")
available = converter.formats
```

Its constructor accepts `strict`, `limits`, and `preserve_facets`. Its `convert` method accepts the same call-specific arguments as the top-level function and uses the converter's policy unless that argument is explicitly overridden. `overwrite` remains call-specific and defaults to false; it is not a persistent converter policy because accidental reuse would be costly.

The top-level `convert` function is behaviorally equivalent to a fresh default `Converter` but may use a private immutable singleton. There is no public mutable default converter, configuration registry, or environment-driven policy.

130.6. Formats and capability discovery

`zenfmt.formats()` and `Converter.formats` return a tuple of immutable `Format` values produced from native capability metadata. Each value includes:

- canonical format id;
- stable plugin id;
- declared extensions in priority order;
- whether the format can be read, written, or both;
- the primary output extension when writable;
- whether the reader needs seekable input;
- whether the writer emits UTF-8 text or arbitrary bytes.

The tuple is deterministic and ordered by the native bundle registry. No hard-coded Python list is permitted. The first release therefore reports all nineteen existing readers and the Markdown writer without claiming a writer that is not compiled into the release.

Capability metadata also drives validation, alias resolution, artifact naming, and the `Conversion.text` property. This prevents four slightly different format registries from appearing in the CLI, engine, Python code, and docs.

130.7. Limits and strictness

`Limits` is a frozen, slotted data class with one positive integer field for every field in the engine's public `Limits`. Field names and defaults match Zig exactly, including hard caps. Unknown keywords fail at Python construction; zero, negative, Boolean, non-integer, and hard-cap-violating values raise `ValueError` before native work. The native engine validates them again because the ABI is a trust boundary.

Adding a native limit is an additive Python change only when the same change adds its typed Python field and the native/Python parity test agrees on name, width, default, and hard cap. Removing, renaming, or changing the meaning of a limit is a breaking API change and an amendment to the engine record. This record adds `max_output_bytes` because artifact output is not otherwise directly bounded; its 512 MiB default matches `max_input_bytes` and is checked at the shared artifact sink during writer emission, before another byte is accepted, in every output mode. A breach fails the conversion with the new limit-class report `core.output-too-large`. This record is the design record documenting that limit; the book's reference chapter gains matching limit and report rows as part of delivery, and ZDS 0013's limits table remains authoritative for the limits it lists.

`Strictness` is a `str` enum with `OFF`, `CONTENT`, `STRUCTURE`, and `EXACT`. Successful reports are not promoted by Python; the selected grade is passed to the native planner so refusal happens before output exactly as ZDS 0013 requires.

130.8. Successful result

`Conversion` is a frozen, slotted value with these public properties:

Property	Contract
`content`	Artifact `bytes` for memory output, otherwise `None`.
`text`	UTF-8 decoded artifact text for an in-memory textual writer. Raises `TypeError` for path output or a binary writer; never guesses an encoding.
`path`	`pathlib.Path` for path output, otherwise `None`. The spelling supplied by the caller is preserved rather than silently resolved.
`name`	The basename used in the manifest and for deterministic resource names.
`source_format`	Canonical reader id selected by the engine.
`output_format`	Canonical writer id selected by the engine.
`resources`	Tuple of `Resource` values, including embedded bytes in memory mode and published paths in path mode.
`manifest`	The typed `Manifest` value and its exact canonical JSON.
`reports`	Tuple of every note and warning in canonical native order.

The result has no `.ok` flag: if a `Conversion` exists, conversion succeeded. Expected failures raise exceptions. This follows ordinary Python practice without throwing away zenfmt's structured failure data.

`Resource` distinguishes `embedded` and `external` resources. An embedded resource has a relative artifact path, media type, BLAKE3 digest, source description, dimensions and alt text when known, plus either `content` in memory mode or `path` in path mode. An external resource exposes its unchanged reference and digest scope but is never fetched. Resource collections preserve manifest order.

130.9. Manifest model

`Manifest` retains the exact canonical UTF-8 JSON bytes returned by Zig as `.raw`. It also provides typed access to the schema, source, artifact, reports, document metadata, plugins, media, and facets. Unknown fields are retained. `to_dict()` returns a defensive, JSON-compatible copy for applications that want general traversal or serialization.

The wrapper must not decode and re-encode `.raw`, because doing so could break canonical byte identity or discard unknown preservation fields. Typed model construction validates the required envelope and points to `NativeLibraryError` if the bundled bridge returns an internally inconsistent manifest; malformed user input remains a conversion report, not a bridge error.

130.10. Reports and exceptions

`Report`, `Direction`, and the context variants are frozen, slotted Python models mirroring the native report schema. Stable machine fields remain exact: `code`, `severity`, `loss`, `exit_class`, `count`, contexts, samples, and each direction's optional command and replacement. The engine's canonical manifest and CLI report JSON do not carry `exit_class`; the bridge serializes reports with `exit_class` included through a serializer option, leaving canonical manifest bytes unchanged, and the result-level worst class also crosses the ABI through a dedicated accessor. Human prose may improve between releases, so applications switch on codes and enums rather than matching `str(report)`.

Successful warnings and notes remain in `Conversion.reports`; the library does not print them and does not emit Python warnings. Failed conversion raises this hierarchy:

```
Exception
└── ZenfmtError
    ├── ConversionError
    │   ├── LimitExceededError
    │   ├── UnknownFormatError
    │   └── DestinationExistsError
    ├── InputReadError
    ├── NativeLibraryError
    └── UnsupportedPlatformError
```

`ConversionError` exposes `.reports`, `.primary_report`, `.code`, and `.exit_class`. `InputReadError` represents a failure raised while consuming a caller-owned binary reader before native conversion begins. Its original exception is retained as `__cause__`.

The three subclasses correspond to stable conditions applications commonly handle. All other native conversion failures use `ConversionError`; the library does not create one exception class per report code. Invalid Python arguments continue to use `TypeError` and `ValueError`, but their messages obey the same Elm-style structure. Loader, ABI, corrupt wheel, and impossible native-result conditions use `NativeLibraryError`, never `ConversionError`, because they are installation or library defects rather than document failures.

130.11. Elm-style Python error contract

The native `Report` contract from ZDS 0002 is also the minimum quality bar for errors created by the Python layer. No public operation may leak a bare `ctypes.ArgumentError`, `UnicodeDecodeError`, `JSONDecodeError`, `KeyError`, or unexplained `OSError`. The underlying exception is chained as `__cause__` when it is useful for debugging, while the public exception explains the failure in zenfmt terms.

Every `ZenfmtError` exposes these common details:

Detail	Contract
`.code`	Stable namespaced machine id. Python-originated ids use the `python.` namespace.
`.title`	Short human headline answering what happened.
`.problem`	Plain-language explanation with the offending value or operation when safe.
`.context`	Structured argument, path, platform, package, or native ABI location when available.
`.consequence`	What stopped, what remained untouched, and whether any artifact was produced.
`.directions`	One or more structured `Direction` values giving concrete next actions.
`.hint`	Convenient rendering of the first direction; always present.
`.details`	Read-only structured diagnostic facts. Raw document bytes and secret-bearing environment data are excluded; callers still decide whether paths and names are safe to log.

`str(error)` is a compact, color-free Elm-style renderer containing title, problem, context when useful, consequence, and a final `What you can do:` section listing every direction — the same rendering contract as the engine's text renderer, so one failure reads the same through the CLI and Python. Conversion errors derive these fields and text from the primary native report; the wrapper does not rewrite its facts. Python-originated failures come from a tested diagnostic catalog with stable codes and direction templates, rather than ad hoc strings at each `raise` site.

An invalid argument therefore looks like:

```
INVALID SOURCE TYPE

`source` must be a path, bytes-like object, or binary reader; received `int`.

The conversion did not start, and no output or manifest was created.

What you can do:

    Pass a path such as `Path("report.docx")` or document bytes.
```

A bridge mismatch names both versions, says that conversion did not start, and provides a complete reinstall command as its first direction. A destination failure names the exact destination operation, states that no manifest was published, and uses the native report's safe overwrite or alternate-path direction. "Try again," "invalid value," and "native error" without an actionable direction are test failures, not acceptable messages.

Programmer argument mistakes remain catchable as ordinary `TypeError` or `ValueError`. They carry the same five-part rendered message but are not given stable application-control-flow codes. `KeyboardInterrupt`, `SystemExit`, and `GeneratorExit` are process control signals rather than library failures and are re-raised unchanged. An allocation-exhaustion path uses a preallocated minimal message whose consequence says that no completed result was returned and whose hint recommends reducing the input/output limits or freeing memory.

130.12. Documentation and compatibility

Every public name has a docstring with a minimal example, parameter contract, return type, failure modes, and security-relevant behavior. The package README starts with path, bytes, strict, and path-output examples before explaining installation or internals. The API reference is generated from the shipped annotations and docstrings; examples are run as tests where practical.

The compatibility policy is:

- public names are exactly those exported by `zenfmt.__all__` and documented;
- fields of frozen public models are stable within a major version;
- new optional parameters, enum members, model properties, formats, reports, and limits may be additive minor changes;
- report wording is not stable, while report codes and structured meanings are;
- positional expansion is prevented by keyword-only options;
- removing or changing public behavior requires a major version and a ZDS;
- deprecations warn for at least one minor release before removal when the security impact permits a transition period.

131. Native Boundary

131.1. Why a versioned C ABI loaded by `ctypes`

The bridge uses the platform C calling convention and no CPython API. The Python layer loads it with `ctypes.CDLL`, which releases the GIL for ordinary foreign calls. This gives one native artifact per operating system and architecture rather than one per CPython minor ABI, and keeps Python reference-counting out of the engine.

The public Python package is the only supported consumer of this initial C ABI. The bridge symbols remain private implementation detail even though a C loader can see them. Stabilizing a general C library is a separate design decision.

131.2. ABI shape

The bridge exposes only these categories of operation:

- query ABI and zenfmt release versions;
- retrieve canonical capability JSON;
- convert a path input or byte input to a memory ensemble or path ensemble;
- read status, exit class, selected formats, reports JSON, manifest JSON, artifact bytes, and resource descriptors from an opaque result;
- release exactly one opaque result.

Options cross the boundary as a small versioned UTF-8 JSON object with a deterministic key order and no floating-point values. Large input, artifact, and resource byte sequences cross as pointer-length slices, never base64 inside JSON. Paths carry an explicit native encoding: raw bytes on POSIX or UTF-16 code units on Windows. The ABI uses fixed-width integers, lengths rather than sentinel termination, and numeric status tags whose meaning is fixed by the ABI version. It does not expose Zig structs, enums, allocators, error unions, slices, or alignment assumptions directly.

The result is an opaque handle owned by the bridge. Accessor slices are borrowed and valid until the one release call. Python copies them inside `try/finally` and releases the handle even if JSON decoding or model construction raises. A null result, invalid length, unknown status,

invalid UTF-8 where UTF-8 is required, or inconsistent manifest is a `NativeLibraryError` and a bridge test failure.

No callbacks enter Python during conversion. The bridge does not retain input pointers after the call, store a Python object, invoke Python allocation, or depend on Python thread-local state. This keeps the GIL release safe and makes the call graph one-directional.

131.3. Memory artifact ensemble

The existing writer-stream result is insufficient for a complete Python memory result. Stream output already returns the canonical manifest, but embedded resources are projected only for path output: the media plan never runs, so the manifest carries no media entries and image targets keep reader-internal names. The bridge therefore requires a narrow engine addition: a memory publication sink that performs the same deterministic resource naming, target rewriting, digests, and manifest construction as path publication, but returns artifact and resource bytes instead of opening final paths, and populates the result only after the manifest is complete.

Concretely, the engine's output specification gains a memory variant carrying the artifact basename supplied by the wrapper, and the conversion result gains an optional memory ensemble — artifact bytes plus embedded resource files — and the selected canonical reader and writer ids. Failure paths never populate the ensemble. Writer descriptors additionally declare whether they emit UTF-8 text or arbitrary bytes so capability metadata can report it.

This is an engine capability, not Python-specific document logic. Its native tests compare memory and path publication: artifact bytes, relative resource names, digests, reports, and manifest semantic content must match, with only the intentionally different destination representation allowed. Allocation exhaustion during a bridge conversion follows the engine's canonical contract: a failed conversion carrying exactly one `core.out-of-memory` report, as pinned by the repository's allocation-failure suite.

131.4. Loading and version checks

The wheel stores one bridge under a private package resource directory. The canonical bridge filenames are `libzenfmt_py.so`, `libzenfmt_py.dylib`, and `zenfmt_py.dll`. The loader selects the exact filename for the running platform and opens its absolute package path. It never searches the current directory, `PATH`, a system library directory, a user configuration location, or an environment override. This prevents library preloading and version-confusion surprises.

Before the first real operation the loader verifies:

- the bridge ABI major equals the Python layer's required ABI major;
- the bridge ABI minor is at least the minimum understood minor;
- the native zenfmt version equals the installed distribution version after the documented SemVer-to-PEP-440 mapping;
- capability JSON has the supported schema version;
- the bridge reports the expected pointer width and native path convention.

Failure names the detected platform, installed Python version, wheel version, native version when readable, and a safe reinstall command. It does not fall back to another library. For development installs, the Hatchling hook is a no-op for editable builds; `zig build python-sync` builds the host bridge and stages it from `zig-out` into the source-layout package's private resource directory, which the editable install exposes as the environment's package-resource location. The runtime loader uses the same resource rule as a wheel and never searches `zig-out` itself.

131.5. Concurrency and process behavior

The bridge owns no mutable process-wide conversion state. Capability metadata may be cached after validation; conversion arenas and result handles are per-call. `Converter` values are immutable, so the same instance may be called from multiple Python threads. `ctypes.CDLL` releases the GIL for the native call and reacquires it before Python model construction.

Forking a multithreaded process while a conversion is active inherits the usual Python and operating-system hazards. The supported rule is to create worker processes before starting conversion threads or use the `spawn` start method. The library registers no background threads, signal handlers, `atexit` hooks, or fork hooks.

132. Project and Build Implementation

132.1. Repository layout

The repository root is a pure Zig project; the Python distribution is a self-contained uv project below `python/`, so language surfaces publish independently and the root carries no Python configuration:

```
zenfmt/
├── build.zig              # the monorepo orchestration graph, Python steps
included
├── build.zig.zon          # canonical release version and Zig dependency lock
├── python/                # the complete uv/Hatchling project
│   ├── pyproject.toml     # PEP 517/621, uv, Ruff, and pytest configuration
│   ├── uv.lock            # committed Python resolution
│   ├── hatch_build.py     # packaging adapter; delegates native builds to Zig
│   ├── README.md          # PyPI-facing readme
│   ├── LICENSE            # MIT text carried into wheel and sdist metadata
│   ├── src/zenfmt/        # public package and private ctypes adapter
│   │   ├── py.typed
│   │   └── _native/       # wheel-only packaged bridge artifact
│   ├── tests/             # pytest unit, integration, and contract tests
│   └── benchmarks/
│       └── python_api.py  # installed-wheel cold/warm API benchmark worker
├── bindings/python/       # Zig bridge source and ABI contract tests
├── core/
├── support/
├── formats/
├── src/
├── cli/
├── benchmarks/
│   ├── benchmark.zig      # unified child-process comparison harness
│   └── results/           # generated JSON and Markdown result files
└── docs/
```

`python/src` is a source layout, so tests exercise an installed package rather than accidentally importing repository files. The bridge is under `bindings` because it is a language boundary; the public Python implementation remains under `python`. The source distribution stays standalone-buildable because the sdist target force-includes the Zig sources it needs from the repository at build time (see the source-distribution section); nothing is vendored in git.

`python/pyproject.toml` records `Vikrant Rathore` as the project author by name. Project documentation credits assistance from `Ronak Rathore`. The package uses the repository's MIT license;

`python/LICENSE` is a copy of the root license text so wheel and sdist metadata carry it, with the root file staying canonical.

132.2. Division of tool authority

Tool	Owns
`zig build`	The monorepo graph, bridge compilation, target and optimize mode, native tests, artifact placement, and aggregate developer gates.
`uv`	Python interpreter/environment selection, dependency groups, `uv.lock`, command execution, PEP 517 frontend behavior, building, clean install checks, and publication.
Hatchling	PEP 517 metadata, source-layout inclusion, native-artifact inclusion, platform wheel tags, source distribution contents, and the thin hook that delegates native compilation back to `zig build`.
Ruff	All Python linting, import sorting, and formatting.
pytest	All Python unit, integration, API contract, and installed-artifact tests.

uv is the project manager, not the native compiler. Hatchling is selected because uv's own build backend is intentionally limited to pure-Python projects; uv remains the build frontend and environment manager for any PEP 517 backend.

Runtime dependencies are empty. A `dev` dependency group contains pytest and Ruff, locked in `uv.lock`. Build requirements are constrained to compatible Hatchling versions and participate in release build constraints. The lockfile is changed only through uv and committed with dependency changes.

132.3. Root build steps

The root graph adds these explicit steps and folds their checks into existing aggregates:

Command	Contract
`zig build python-native`	Build the host bridge into `zig-out` without invoking Python.
`zig build python-sync`	Build the host bridge and run uv's locked development-environment sync, staging the bridge into that environment.
`zig build python-test`	Build the host bridge and run pytest through uv.
`zig build python-lint`	Run `ruff check` through uv.
`zig build python-format`	Run Ruff's formatter in write mode.
`zig build python-format-check`	Check Ruff formatting without edits.
`zig build python-wheel`	Build the host platform wheel through uv after the native artifact is available.
`zig build benchmark-python`	Build and clean-install the host wheel, then run the cold, warm, memory/path, error, and concurrency API profiles.

Command	Contract
`zig build benchmark`	Run the unified corpus comparison, including the installed zenfmt wheel as well as the native CLI, pandoc, and anydoc.
`zig build python-check`	Run native ABI tests, lint, format check, pytest, wheel inspection, and a clean-install smoke test.

`zig build test` depends on `python-test`; `zig build fmt-check` depends on `python-lint` and `python-format-check`; the existing `benchmark` step depends on `benchmark-python`; and the release gate depends on `python-check` plus a recorded benchmark run. No aggregate check or release gate exists before this record; the `python-check` aggregate, the wheel workflow, and the tag-driven release workflow are new constructs delivered here, not extensions of existing ones. `python-native` honors the standard `--prefix` flag so one host can cross-compile the bridge for every wheel target into per-target directories without artifact collisions. Missing uv is a clear prerequisite failure, never a silent skip. Direct uv/Ruff/pytest commands remain documented for Python-focused contributors, but CI calls the root Zig steps so the monorepo has one graph.

The build hook does not duplicate target selection or run arbitrary shell strings. It invokes a named root Zig step with explicit arguments, requires the expected output at an exact path, and then gives that artifact to Hatchling. Wheel metadata derives its platform tag from the same validated target tuple.

132.4. Repository integration obligations

Integrating a new language and a bridge into the monorepo touches several existing contracts that are easy to miss:

- `build.zig.zon` gains `.paths` entries for `bindings` and `python` (the latter covers the whole Python subproject, `pyproject.toml`, `hatch_build.py`, and `uv.lock` included), and its `.version` becomes the canonical `0.1.0`; `build.zig` reads and validates that version and embeds it in the CLI and the bridge.
- The root format check and end-to-end test lists are explicit arrays; the bridge sources join the format list and the new engine test files join the test list.
- The documentation-drift gate requires every new engine report code and every new limit to appear in the book's reference chapter; `core.output-too-large` and `max_output_bytes` therefore make that chapter a deliverable. Bridge `bridge.*` codes and Python `python.*` codes live outside the scanned engine roots and are deliberately absent from that catalog because the ABI is private.
- The allocation-failure suite pins the default bundle at nineteen readers and the single `core.out-of-memory` report; the bridge must preserve both.
- The benchmark harness's tool set and the book's benchmark chapter hard-code the compared tools; both are named deliverables of the benchmark phase.

132.5. Python configuration

`python/pyproject.toml` is the only Python tool configuration file. It contains:

- PEP 621 project metadata with `requires-python >= 3.10`, MIT licensing, project URLs, classifiers, and name-only authorship;
- a dynamic version derived from `build.zig.zon`;
- Hatchling build-system and custom hook configuration;
- uv's `dev` dependency group and supported-environment constraints;

- Ruff target version, lint selection, formatter policy, and package source roots;
- pytest paths, strict markers/config, and `--import-mode=importlib`.

Ruff is configured once for `python/src`, `python/tests`, `python/benchmarks/python_api.py`, and `python/hatch_build.py` — every Python file in the repository lives inside the subproject, so one root-relative configuration covers them all. The formatter's output is canonical; contributor style arguments are not re-litigated in reviews. Lint selections emphasize correctness, security-prone APIs, modern Python, import hygiene, annotations, and tests, with every ignore narrowly documented beside the configuration.

133. Packaging and Release

133.1. One release version

`build.zig.zon` remains the canonical SemVer version for the monorepo, and it becomes `0.1.0` with this record's delivery. `build.zig` reads the value, validates it, and embeds it in the CLI's `--version` output and the bridge's version query — none of which existed before this record. The Python metadata hook translates the same value deterministically to PEP 440, and `zenfmt.__version__` reads installed distribution metadata. Stable versions map unchanged; pre-release identifiers use the corresponding PEP 440 spelling; development or local build identifiers are never uploaded as a stable public release.

The bridge embeds the same canonical version. Build and import checks reject a Python/native mismatch. A release tag, CLI version, Zig package version, Python distribution version, and bridge version therefore identify one source state.

133.2. Wheels

Each wheel contains platform-independent typed Python source plus one native bridge, and carries a platform tag. Because the bridge uses no Python C ABI, the Python tag is `py3`, the ABI tag is `none`, and `Requires-Python` enforces the language floor. Wheels are never marked `any`.

The initial required matrix is:

Platform family	Required architectures/baseline
Linux glibc	`manylinux_2_17` on `x86_64` and `aarch64`.
Linux musl	`musllinux_1_2` on `x86_64` and `aarch64`.
macOS	macOS 12 or later on `x86_64` and `arm64`.
Windows	64-bit `x86_64` (`win_amd64`).

The Python layer targets Python 3.10 and later and uses only standard-library APIs available at that floor. Release CI tests every supported CPython minor from the declared floor through the latest stable release. PyPy support is a deferred post-`0.1.0` additive gate rather than a first-release requirement, subject to the same `ctypes` and platform-tag contract. A platform is listed as supported only while its wheel is built, inspected, installed, loaded, and exercised on that platform. Additional platforms are additive releases after the same gate.

Wheel inspection must confirm:

- one and only one expected native bridge;
- no object, test, cache, credential, temporary, or developer path leakage;
- correct `Root-Is-Purelib`, Python, ABI, and platform tags;
- the root license and typed marker are present;

- native dependency inspection shows only permitted baseline system libraries;
- the loader succeeds from a clean environment with a path containing spaces and non-ASCII characters;
- the installed package does not need Zig, uv, Hatchling, Ruff, or pytest.

133.3. Source distribution

The release also publishes one source distribution, built from the `python/` subproject so `pyproject.toml` sits at the archive root as PEP 517 requires. It includes the Python package and tests, build hook, `pyproject.toml`, lock needed for release verification, and the subproject readme and license; the Zig sources required to build the same bridge — `build.zig`/`build.zig.zon`, bridge, engine, support, default format, and umbrella sources — are force-included from the repository at build time and nested under `engine/` inside the archive, which keeps the Zig tree's internal layout intact (its `src/` never collides with the Python `src/` layout) without vendoring a second source tree in git. The packaging hook resolves the Zig build root in either place: the sdist-local `engine/` directory or the repository parent of `python/`. The sdist excludes repository-only docs output, benchmarks corpora, temporary files, and unrelated generated artifacts.

Building from the source distribution requires the minimum Zig version named by `build.zig.zon` and a supported Python. The PEP 517 failure for a missing or wrong Zig executable states exactly which version is required and recommends installing an official wheel when one exists. The sdist is tested by unpacking it outside the checkout, building a wheel through uv with local-source overrides disabled, installing that wheel into a clean environment, and running the public smoke suite.

An sdist that depends on parent-directory monorepo files is invalid. The release gate inspects its file list and builds only from the archive.

133.4. PyPI publication

The intended PyPI project name is `zenfmt`. Publication occurs only from the tag-driven release workflow's protected `pypi` environment after all wheel and sdist jobs complete. Trusted Publishing with short-lived OIDC credentials is preferred once the project publisher is configured. The bootstrap release may instead use a project-scoped PyPI API token stored only as that environment's encrypted `PYPI_API_TOKEN` secret. The credential never enters a build job, repository file, artifact, command output, or maintainer environment file.

The publication job downloads the artifacts built and exercised by unprivileged jobs, checks that filenames and embedded versions form one complete matrix, and publishes those exact bytes through the PyPA publish action with PEP 740 attestation generation enabled. A TestPyPI release candidate is required when TestPyPI authority has been configured and whenever packaging machinery changes materially after the bootstrap release. The same tag builds standalone CLI archives for every wheel target and creates one GitHub release containing the CLI archives, wheels, sdist, and SHA-256 checksum manifest.

No release job builds after it receives publication authority. Build jobs do not receive publication authority. Published files are immutable; a broken artifact is fixed with a new version rather than replaced.

134. Testing and Quality Gates

134.1. Python unit tests

pytest unit tests cover the Python layer without requiring real conversion for every case. A private fake-bridge fixture supplies valid and invalid capability, result, report, manifest, and resource payloads. Tests cover:

- every accepted and rejected source type;
- binary-reader chunking, exact limit boundaries, non-bytes reads, and ownership/close behavior;
- format alias normalization and ambiguity;
- `Limits` type, range, Boolean, unknown-field, and hard-cap validation;
- strictness normalization;
- immutable/hash and copy semantics of public models;
- manifest unknown-field preservation and exact `.raw` bytes;
- exception selection, attributes, messages, and chaining;
- the Elm-style four-question contract, stable Python diagnostic codes, non-empty `What you can do:` rendering, and concrete directions for every cataloged failure;
- proof that no expected path leaks raw FFI, decoder, lookup, or filesystem exceptions through the public API;
- result-handle release on success and on every Python exception path;
- lazy loading and deterministic loader failures;
- public exports, signatures, annotations, doc examples, and `py.typed`.

Tests assert behavior, not private module layout. Private FFI definitions also have size, signedness, calling-convention, and symbol-name tests so an ABI drift fails close to its cause.

134.2. Native and integration tests

Zig ABI tests call the bridge as a C consumer would: valid paths and bytes, zero lengths, maximum lengths, invalid tags, invalid UTF-8, embedded NUL, misaligned/nullable inputs where permitted, repeated access, and single release. The bridge is fuzzed at its request/options boundary independently of Python.

pytest integration tests use the real host bridge and the repository corpus. They compare Python conversion with direct Zig conversion for:

- every registered reader into every registered writer;
- explicit and detected formats;
- bytes, binary readers, Unicode paths, and path-like objects;
- default and overridden limits;
- every strictness grade and representative loss;
- successful reports and failed report arrays;
- stale, malformed, valid, and absent adjacent manifests;
- embedded and external resources in memory and path modes;
- overwrite refusal and publication failure;
- canonical artifact, resource, digest, and manifest bytes;
- simultaneous calls through one `Converter` from multiple threads.

No test silently skips a format compiled into the default bundle. Capability metadata parameterizes the matrix; an unexpected new format creates required cases automatically.

134.3. Installed-artifact tests

Tests against the checkout are necessary but insufficient. Every release artifact is installed into a clean environment and must pass:

- import, version, ABI, and capability queries;
- a byte-to-byte conversion;
- a path-to-path conversion with manifest and media validation;
- a structured failure and a limit refusal;
- a concurrent conversion;
- import and conversion with the checkout absent from `sys.path`;
- operation with a read-only installed package directory.

The wheel smoke test runs without Zig and without development dependencies. The sdist smoke test runs from its unpacked source only. A source tree test may not stand in for either one.

134.4. Benchmarking the installed wheel

The repository's benchmark is extended to measure the Python API that users actually install. It must never import `python/src/zenfmt` directly or load a bridge from the checkout. `zig build benchmark-python` builds a ReleaseSafe wheel, installs that exact wheel into a clean isolated uv environment, and runs `python/benchmarks/python_api.py` with the checkout's Python sources absent from `sys.path`. Before timing, the worker records and verifies the wheel filename, wheel SHA-256 digest, installed `zenfmt.__version__`, package location, native version, ABI version, Python implementation/version, and platform tag.

The ordinary `zig build benchmark -Doptimize=ReleaseSafe` step depends on this installed-wheel benchmark and adds `zenfmt-python-wheel` to the existing corpus comparison beside the native `zenfmt` CLI, pandoc, and anydoc. The wheel build and installation are setup costs and are reported as release metadata, not included in conversion latency.

134.4.1. Benchmark profiles

Cold and warm results answer different questions and must never be merged into one headline number:

Profile	Timed region	Question answered
cold import	A fresh interpreter imports `zenfmt`, validates distribution metadata, and reads `__version__`.	What does adding the package to a short-lived Python program cost?
cold first conversion	A fresh interpreter imports the installed package, loads/verifies the bridge, constructs a default `Converter`, converts one corpus path to memory, and materializes result models.	What does a one-shot script or job pay?
warm path → memory	One imported process and `Converter` convert each corpus path into artifact/resource bytes and Python models.	What does the normal long-running service API cost?

Profile	Timed region	Question answered
warm bytes → memory	Source bytes are loaded before timing; the call includes FFI validation, conversion, native-to-Python copies, manifest and report decoding, and model construction.	What is Python/FFI overhead when application I/O is already complete?
path publication	The installed API converts a source path into a fresh temporary output directory, including artifact, media, and manifest publication. Cleanup occurs after timing.	Does the Python entry point preserve CLI-grade end-to-end path performance?
diagnostic failure	Representative unknown-format, limit, and destination failures build structured exceptions and render their Elm-style text.	What does the complete Python error experience cost?
concurrent throughput	One immutable `Converter` processes independent corpus documents with 1, 2, 4, and 8 worker threads.	Does GIL release translate into useful parallel throughput without changing results?

The existing real-document corpus is the primary workload. A fixed tiny text-to-Markdown case is added only as a boundary microbenchmark so native loading, validation, copying, and model construction are visible when parsing work is negligible. Synthetic microbenchmark results are reported separately and never used to claim corpus throughput.

134.4.2. Measurement rules

- The default is one discarded warm-up followed by five measured iterations, matching the existing harness. Results report the median; raw samples remain in machine-readable output.
- Cold profiles use a new child process for every sample. The Zig parent measures wall clock, user-plus-system CPU, and peak RSS through the same process/rusage mechanism used for the CLI comparison.
- Warm profiles run repeated calls in one installed-wheel process. Python's monotonic high-resolution and process CPU clocks measure each call; the parent still records whole-worker peak RSS. Garbage collection and temporary output cleanup occur between samples, outside the timed region, without disabling normal runtime behavior inside a call.
- Bytes input is read before the bytes-to-memory timer. Path input opening, reading, and adjacent-manifest behavior remain inside path profile timers.
- Result construction is inside the timed region. The worker consumes artifact length, manifest bytes, report codes, and resource digests before completing a sample so the full public API result is exercised.
- Python wheel and native baselines use the same source revision, optimize mode, limits, formats, corpus files, CPU affinity policy, and iteration count. Environment differences are written to the results rather than normalized away silently.

- The publishable comparison uses the latest stable CPython on the repository's designated benchmark host. Other Python versions and wheel platforms may emit diagnostic result files, but results from different machines are never combined into one aggregate or ratio.
- The benchmark makes no network request. pandoc and anydoc dependencies and the document corpus are prepared before timing using the repository's existing workflow.

The benchmark reports raw values and ratios; it does not subtract the native median from the Python median, because independently sampled differences are too noisy to call "binding overhead." The cold installed-wheel row is compared with the CLI's child-process row. Warm Python memory calls are compared with an equivalent in-process Zig memory-publication profile added to `benchmarks/stages.zig`. Ratios always name numerator and denominator, and a larger-is-better throughput ratio is never placed beside a smaller-is-better latency ratio without an explicit label.

134.4.3. Correctness before timing

Performance data is publishable only after parity checks pass. For every shared corpus case, the installed wheel and same-revision native path must agree on:

- selected input and output format ids;
- artifact bytes and digest;
- embedded resource bytes, relative names, order, and digests;
- canonical report codes, severities, counts, losses, and directions;
- manifest semantic content, allowing only the destination-name fields that intentionally differ between memory and path profiles;
- success/failure classification and strict/limit refusal behavior.

A mismatch marks the sample `ok=false`, records the first bounded difference, excludes the sample from aggregates, and fails the benchmark step after writing diagnostic results. Unsupported formats remain visible as `supported=false`; they are never silently removed from the denominator.

134.4.4. Result artifacts and reporting

`benchmarks/results/python.json` is the detailed installed-wheel result file. It contains schema version, source revision, wheel identity/digest, interpreter, platform, native ABI/version, benchmark configuration, raw samples, medians, parity status, and concurrency throughput. The existing `benchmarks/results/latest.json` gains the cold `zenfmt-python-wheel` tool row and a reference to the detailed Python result schema.

`benchmarks/results/results.md` remains generated human-readable output and gains a **Python wheel API** section, rendered by the harness from the detailed Python results, showing cold import, first call, warm memory, path publication, error rendering, concurrency, and native comparison ratios. The benchmark harness and the book's benchmark chapter both enumerate the compared tools explicitly, so adding the wheel is a code change in each: the harness gains a fourth tool and the chapter gains the fourth tool name, palette entry, adjusted bar layout, and a **Python wheel API** section reading the detailed Python results. Once extended, the chapter reads generated JSON only, so published documentation stays generated. Result files are produced by the harness and never edited by hand.

The initial implementation records measurements rather than inventing a performance budget. After representative baselines exist, a regression budget may be adopted by amendment with a named profile, corpus, statistic, and machine policy. Review must not hide a slow cold start behind warm throughput or use the tiny boundary case as a marketing result.

134.5. Acceptance gates

This ZDS may move to `committed` only when:

- the documented public surface exists with no undocumented public aliases;
- all root Zig and Python quality steps pass;
- every Python and native failure exposed by the public API satisfies the Elm-style contract and provides at least one actionable direction rendered under `What you can do:`;
- the ABI is versioned, documented privately, fuzzed, and checked at load;
- memory output returns resources and is parity-tested with path output;
- `max_output_bytes` is enforced during writer emission in every output mode;
- the default format table comes only from native capability metadata;
- failure paths leak no result handle and publish no path manifest;
- a wheel for every required target passes inspection and clean-install tests;
- the sdist builds a passing wheel outside the monorepo checkout;
- the installed-wheel benchmark covers every required profile, passes artifact parity, and writes complete wheel/version/platform metadata to generated results;
- the unified `zig build benchmark` report contains the Python wheel beside the native CLI and keeps cold, warm, and concurrent measurements distinct;
- release version parity is mechanically enforced;
- user documentation clearly states input authority, adjacent-manifest loading, resource limits, external-reference behavior, and in-process native risk;
- PyPI publication uses the protected environment, a project-scoped credential (Trusted Publishing when configured), and artifact attestations;
- the GitHub release contains the complete wheel matrix, sdist, CLI matrix, and checksum manifest.

135. Security Considerations

135.1. Threat model

The engine threat model remains: an attacker supplies a document, and possibly many documents, to a Python process converting on someone else's behalf. The Python boundary adds malicious Python objects, hostile paths, corrupt package artifacts, ABI mismatch, native library preloading, excessive buffer copying, and accidental authority through convenience behavior.

135.2. Input authority

Only an explicit path source grants filesystem read authority. Bytes and binary readers never cause the library to open their display name. Path input retains the engine's one adjacent-manifest probe and digest verification; the public docs state this because it is additional filesystem input derived from the selected path. No input causes network access, external entity expansion, include following, link fetching, or plugin discovery.

Python protocol calls are minimized. The wrapper calls `os.fspath` once for a path-like value or `read` repeatedly for a recognized binary reader; it does not inspect arbitrary attributes except the optional display-only `.name`. Ordinary exceptions from those calls become an Elm-style `InputReadError` or argument error with the original value chained as `__cause__`; process-control exceptions are re-raised unchanged. Every path releases any native state.

135.3. Resource exhaustion and copies

Native limits remain authoritative and are validated twice. Binary-reader ingestion is bounded before accumulating the full input. Integer conversion is checked before narrowing to fixed-width

ABI fields. Buffer sizes use checked arithmetic, and a reported native length is validated against the relevant limit before Python copies it.

Memory conversion necessarily owns artifact and embedded resource bytes. `max_output_bytes` bounds the artifact while `max_resource_bytes` bounds embedded resources; checked addition prevents the ensemble or a Python copy from wrapping an integer. The documentation directs services handling large or untrusted output to explicit path conversion inside a quota-controlled directory. The bridge never returns a Python view onto freed native memory.

135.4. Native library loading and supply chain

`ctypes` bypasses Python's memory safety, so its use is isolated in one private module with complete prototypes and tests. The package loads an absolute, wheel-owned artifact only, checks ABI and version before conversion, and has no system fallback or runtime download. Wheel records, platform tags, native dependency inspection, release attestations, and clean-install tests defend the packaging boundary.

Python build and development dependencies are locked. Release builds use isolated PEP 517 environments with explicit build constraints and hash verification where supported. Trusted Publishing avoids a long-lived upload credential; when a bootstrap token is used, it is project-scoped and isolated to the protected publication environment. Neither mechanism proves code safety; branch, tag, workflow, and approval protections remain required.

135.5. Filesystem output

`overwrite=False` is the shortest path. Path publication continues to use unpredictable same-directory temporary names, bounded relative media paths, digest-bound manifests, and manifest-last commit. The wrapper does not pre-check existence and then open files itself, which would introduce a race. It passes the decision to the engine and returns the engine report.

Callers remain responsible for choosing an output directory whose permissions, quotas, symlink policy, and trust boundary match their application. The library does not claim that atomic rename is a sandbox. Service documentation should recommend a per-job directory with restrictive permissions.

135.6. In-process execution

The bridge runs in the Python process with that process's authority. A memory safety defect or invariant panic can terminate or compromise the host. The native engine's ReleaseSafe, bounded, non-recursive, and adversarial-test requirements therefore apply unchanged. The docs state that process isolation is an application architecture choice for higher-risk workloads; the first release does not imply a sandbox because the API looks high level.

136. Operational Considerations

The runtime library has no daemon, cache directory, configuration file, telemetry, updater, network request, or background thread. Installing a wheel adds the Python package and one bridge; uninstalling it removes both.

Cold first use pays shared-library loading and capability validation once per process. Later calls reuse the loaded handle and immutable capability models. Conversions remain independent. Benchmarks record import time, first-call overhead, steady-state overhead, artifact throughput, and peak memory for path and memory modes. Targets must be recorded in benchmark result files before being quoted as promises.

Logging belongs to the application. The library prints nothing to stdout or stderr and does not configure Python logging. Applications may render returned reports into their own logs, UI, HTTP response, or telemetry with stable codes.

Support reports should include Python version, platform tag, `zenfmt` version, native version/ABI when loadable, source and output format ids, and report codes. They should not require users to attach private documents or manifests containing document metadata. The exception renderer excludes absolute native library build paths and developer-machine details.

ABI major changes require a coordinated Python/native major release. ABI minor changes are append-only and capability-gated. The loader error for mixed files is intentional; silently continuing would turn packaging corruption into data corruption.

137. Delivery Plan

137.1. Phase 1: native contract

- Add and enforce `max_output_bytes` at the shared artifact sink for every output mode, with the `core.output-too-large` report and matching reference chapter rows.
- Add memory artifact/resource publication to the engine, with the memory output variant, result ensemble, and selected-format ids.
- Add the bridge-facing report serializer option carrying `exit_class` and the writer text/binary emission declaration.
- Plumb the canonical `build.zig.zon` version, bumped to `0.1.0`, into the build graph, CLI, and bridge.
- Specify the private ABI schema and symbol contract in the bridge directory.
- Build the bridge through a root Zig step.
- Add native ABI, ownership, adversarial, and memory/path parity tests.

137.2. Phase 2: Python surface

- Add project metadata, uv lock, source layout, and public typed models.
- Implement validation, capability discovery, conversion, results, and exceptions in the order presented here.
- Implement the shared Elm-style exception renderer and Python diagnostic catalog before adding individual failure sites.
- Add Ruff and pytest configuration and fold them into root checks.
- Write the quickstart and API reference alongside tests of their examples.

137.3. Phase 3: packaging and benchmark

- Add the Hatchling metadata/build adapter that delegates to root Zig steps.
- Produce and inspect a host wheel and standalone source distribution.
- Add the required cross-platform wheel matrix and clean-install tests.
- Establish version parity and reproducibility comparisons.
- Extend the existing benchmark harness with the installed-wheel worker, equivalent in-process Zig memory baseline, parity checks, result schemas, and generated report section specified above.
- Extend the benchmark harness's tool set and the book's benchmark chapter with the wheel row and the **Python wheel API** section.
- Record a full ReleaseSafe corpus run from the clean-installed wheel and review cold, warm, path, error, memory, and concurrency results before publication.

137.4. Phase 4: publication

- Reserve the PyPI project and configure the protected publication environment; use a project-scoped bootstrap token until its trusted publisher is active.
- Add the wheel-matrix and tag-driven release workflows.
- Rehearse a release candidate on TestPyPI when that publisher is configured.
- Generate attestations, publish the complete `0.1.0` matrix, and run post-publish install tests against PyPI.
- Attach the complete wheel, sdist, CLI, and checksum set to the immutable `v0.1.0` GitHub release.
- Move this record through accepted, published, and committed states only as the corresponding lifecycle conditions are met.

138. Alternatives Considered

138.1. Shelling out to the CLI

Rejected as the primary backend. It offers crash isolation and may remain a documented application choice, but it adds process startup, executable lookup, pipe/temp-file complexity, media handling, and duplicated option/error contracts. A high-quality local Python library should not make users parse its own CLI.

138.2. A CPython extension using the limited API

Rejected for the first release. An `abi3` extension could produce efficient objects directly, but it makes Python reference ownership, exception state, interpreter lifecycle, and C API compatibility part of the native engine boundary. The conversion call is coarse enough that `ctypes` marshalling is not the performance center. The C ABI can later receive a separate CPython adapter without changing the public Python API.

138.3. cffi

Rejected because it adds a runtime or build dependency without solving a problem the narrow fixed C ABI needs. `ctypes` is in the supported Python standard library, supports explicit prototypes, loads ordinary shared libraries, and releases the GIL for the chosen calling convention.

138.4. A pure-Python reimplementation

Rejected. It would duplicate nineteen parsers, writer lowering, canonical manifest behavior, diagnostics, limits, security fixes, and tests, then drift from the Zig product. The Python library is an adapter, not a second converter.

138.5. A remote service client

Rejected as the meaning of the `zenfmt` package. A hosted service has authentication, retries, billing, privacy, availability, and protocol versioning concerns absent from this local library. A future service client should use a distinct package or namespace and its own design record.

138.6. The uv build backend

Not selected because it supports pure-Python projects, while these wheels contain a platform native library and require a build hook plus platform tag control. uv still owns project management and invokes Hatchling through the standard PEP 517 frontend.

138.7. The repository root as the uv project root

Rejected. An earlier revision of this record placed `pyproject.toml`, `uv.lock`, and the packaging hook at the repository root on the theory that a source distribution could not otherwise contain

the Zig engine without vendoring it. That made the whole monorepo double as the Python project: Python tooling configuration polluted a Zig-first root, and publishing language surfaces separately became awkward as each new surface would compete for the same root. The objection to the isolated layout was wrong in fact — Hatchling's sdist force-include maps repository sources into the archive at build time (nested under `engine/`), so a self-contained `python/` project still produces a standalone sdist with no second committed source tree. The adopted layout keeps the root purely Zig, `zig build` in charge of orchestration, and the Python project publishable on its own.

138.8. Wheel-only publication

Rejected as the steady-state release. Wheels give the intended install experience, but a buildable source distribution is important for inspection, archival, unsupported environments, and downstream packagers. The sdist must be genuinely standalone; publishing a knowingly broken sdist would be worse than delaying it.

138.9. Returning status instead of raising

Rejected at the Python surface. The Zig API returns a status because Zig error unions cannot carry its full diagnostic result conveniently. Python exceptions naturally carry typed attributes and compose with application control flow. Successful warnings remain ordinary result data, so exceptions do not erase loss information.

138.10. Mutable global configuration

Rejected. Stripe-like global settings can be convenient for a network client, but conversion policy in a threaded document service must be explicit and test-isolated. Immutable `Converter` values provide reusable defaults without ambient state.

139. Open Questions

No semantic or API question is intentionally left open. Two external facts gate the release phase rather than this record's lifecycle: that the `zenfmt` PyPI project can be placed under project control, and that continuous integration capacity exists — including an arm64 Linux runner with a musl container — to test every required wheel target. If either fact is false, the distribution name or initial support matrix must be amended explicitly rather than weakened silently during implementation.

140. Acknowledgements

Vikrant Rathore authored this implementation specification with assistance from Ronak Rathore.

141. References

- ZDS 0001, **The Zen Discussion Process**.
- ZDS 0002, **zenfmt: Architecture and Implementation**.
- ZDS 0013, **Layered Document IR and Writer Lowering**.
- uv project management.
- uv distribution builds.
- uv package build and publication guide.
- uv build-backend scope.
- Hatch custom build hooks.
- Python platform compatibility tags.
- Python project metadata specification.
- Python `ctypes` documentation.

- Ruff configuration.
- pytest integration practices.
- PyPI Trusted Publishing.
- PyPI digital attestations.