Idiomatic Safe Rust as a Translation Target: Ownership, Lifetimes, and ABI-Compatible Parser Design

A Case Study in `safe-libyaml`

Matthew Long
The YonedaAI Collaboration, YonedaAI Research Collective
Chicago, IL
`matthew@yonedaai.com` · `https://yonedaai.com`

April 16, 2026

Abstract

The migration of legacy C libraries to Rust is bottlenecked less by syntax than by the inverse problem of *recovering* the implicit invariants that C never forced its programmers to state. This paper is a *design specification*: it characterises the *target* side of that recovery by enumerating, in concrete and reviewable form, what idiomatic safe Rust should look like for a streaming parser library, and what engineering rules a translation pipeline must follow to hit that target deterministically. We do *not* report performance numbers for an existing automated port; the proposed crate `safe-libyaml` is in-progress and the figures we cite (e.g. "parity with C") are inherited from S. Ask's *manual* port `libyaml-safer` and stated explicitly as engineering targets, not measurements. The contribution is a systems-engineering specification plus the agent-level build plan needed to realise it. We adopt a deliberately lightweight notation for lifetime subtyping ($'a : 'b$), the borrow judgment $\Gamma \vdash e : \tau\langle\rho\rangle$, and the `Result`/`Option` algebra (Section 2). The notation is a *clarity tool*: it lets us state precisely which components of the design are forced by Rust's type system and which are engineering choices. We do not claim any new formal result about Rust itself; the soundness theorems we cite are RustBelt's [1]. The substantive contributions are: (i) a lifetime-parameterised `Event<'a>` enum with `Cow<'a, str>` payloads (Section 4); (ii) a `thiserror`-derived structured error matching libyaml's existing problem categories (Section 5); (iii) the `Vec`/`VecDeque` mapping for libyaml's `STACK_*`/`QUEUE_*` macro family (Section 6); and (iv) a strict no-`unsafe` core paired with a thin `extern "C"` ABI shim `safe-libyaml-sys` (Sections 9–10) that preserves the exact symbol set required by existing C callers and by Rust crates such as `serde_yaml_ng` and `serde_yaml_bw`. Appendix A is supplementary material: a 32-task agent-level build plan (a detailed, automatable task list for the prototype-building agents that the project's pipeline executes) that demonstrates *implementability* of the design but is not itself a research

result. The paper is self-contained; references to other papers in the *Ferrous Bridge* project [12, 13, 14] are pointers, not prerequisites.

Contents

1 Introduction **4**
- 1.1 What "safe and idiomatic" means in this paper . . . 4
- 1.2 `safe-libyaml` as the running example . . . 4
- 1.3 Outline . . . 5

2 Notation: A Lightweight Framework **5**
- 2.1 Lifetimes as regions . . . 5
- 2.2 The borrow judgment . . . 5
- 2.3 The Result and Option algebra . . . 6

3 Ownership and Borrowing for Streaming Parsers **7**
- 3.1 The single-owner buffer . . . 7
- 3.2 The zero-copy `&[u8]` pattern . . . 8

4 Lifetimes for Event-Based APIs **8**
- 4.1 The lifetime-parameterised `Event<'a>` . . . 8
- 4.2 Escape hatches: `OwnedEvent` and the borrow-vs-clone tradeoff . . . 9

5 Error Handling: thiserror vs. anyhow **11**
- 5.1 Library vs. application errors . . . 11
- 5.2 Why we do *not* chain via `Error::source()` . . . 12
- 5.3 The `Result<Event<'a>, Error>` shape . . . 12

6 Memory Management **13**
- 6.1 The `libyaml` macro family . . . 13
- 6.2 Capacity-doubling proposition . . . 14
- 6.3 When `Cow` earns its keep . . . 14
- 6.4 The `BString` alternative and the UTF-8 default . . . 14

7 no_std + alloc **16**
- 7.1 What changes when shipping for embedded . . . 16
- 7.2 The `core::ffi` story . . . 16
- 7.3 Feature flags . . . 16

8 Tagged Enums vs. C Unions **16**
- 8.1 The discriminant . . . 16
- 8.2 The `#[non_exhaustive]` discipline . . . 17

9 Unsafe Boundaries **18**
9.1 Discipline . . . 18
9.2 The `safe-libyaml-sys` split . . . 18
9.3 Interior unsafety as an anti-pattern . . . 19

10 ABI-Compatible C Export **19**
10.1 The drop-in replacement principle . . . 19
10.2 `#[repr(C)]` layout . . . 19
10.3 The `cbindgen` contract . . . 20
10.4 Symbol-level enumeration . . . 20
10.5 Bridging C `FILE*` to Rust `Read` . . . 21
10.6 Why `serde_yaml_ng` matters . . . 22

11 Tooling: clippy, rustfmt, cargo-deny, cargo-audit, cargo-semver-checks **22**

12 Related Work **23**

13 Results **23**

14 Discussion **24**
14.1 Performance: targets, not measurements . . . 24
14.2 Limitations . . . 24
14.3 Place in the broader project (optional reading) . . . 25

15 Conclusion **25**

A Development Plan for Prototype Agents (Scaffold and Idiom Phase) **27**
A.1 The exact public API the agents must target . . . 27
A.1.1 `safe-libyaml::Parser` . . . 27
A.1.2 `safe-libyaml::Event` . . . 27
A.1.3 `safe-libyaml::Emitter` . . . 28
A.1.4 `safe-libyaml-sys::extern "C"` symbols . . . 29
A.2 Workspace layout (target) . . . 30
A.3 Agent task list . . . 30
A.3.1 Agent A2 — Scaffold Builder . . . 31
A.3.2 Agent OSG-to-Rust emitter . . . 33
A.3.3 Agent C4 — Idiom Polisher . . . 34
A.3.4 Crate-layout agent (umbrella) . . . 35
A.4 Module dependency diagram . . . 36
A.5 Reference `lints.toml` . . . 36
A.6 Hand-off to Phase B . . . 37

1 Introduction

1.1 What "safe and idiomatic" means in this paper

Rust's marketing surface — "a systems language without the segfaults" — underdetermines what "safe" means as an engineering target. We take three operational definitions throughout the paper.

Definition 1.1 (Safe Rust)**.** A Rust crate is *safe* if it contains no `unsafe` blocks outside of a thin, separately-versioned FFI boundary, and if its entire test suite passes under *Miri* [2] with zero undefined behaviour reports under the Stacked Borrows aliasing model.

Definition 1.2 (Idiomatic Rust)**.** A Rust crate is *idiomatic* if it (i) replaces every C-style integer error return with a `Result<T,E>`; (ii) replaces every nullable pointer with `Option<T>`; (iii) replaces every tagged union with an `enum` whose exhaustiveness is checked by the compiler; (iv) prefers iterator adaptors over manual index loops; and (v) passes `cargo clippy -- -W clippy::pedantic` with no allowed lints other than those documented in the crate's lint policy.

Definition 1.3 (ABI-compatible)**.** A Rust crate is *ABI-compatible* with a C library L if it exports a set of `extern "C"` functions whose names, calling conventions, and `#[repr(C)]` struct layouts match L's public header file byte-for-byte as detected by `cbindgen` diff and by `nm` symbol enumeration.

The thesis of the paper is a *specification* thesis: all three properties should be simultaneously achievable for a non-trivial streaming parser, and the engineering rules that would make them achievable can be encoded as agent prompts and CI gates. We propose those rules concretely; the empirical verification that the in-progress `safe-libyaml` actually achieves them at parity with the C reference is the responsibility of the companion verification work and is not claimed here as a result.

1.2 `safe-libyaml` as the running example

The `libyaml` C library is a streaming YAML 1.1 parser/emitter of roughly 9,300 lines of source spread across ten files. The `unsafe-libyaml` crate[1] is a mechanical `c2rust` transpilation that retains the original C control flow but expresses it in pervasively-`unsafe` Rust; it has $\approx$ 90M cumulative downloads as the parser backend of `serde_yaml`, `serde_yaml_ng`, `serde_yaml_bw`, and `serde_norway`. The `libyaml-safer` crate[2] is a one-week manual safe port by S. Ask that demonstrates the exercise is feasible by a skilled human, and which we use as a quality and performance reference. Our target crate `safe-libyaml` is the planned *automated* safe port produced by the *Ferrous Bridge* pipeline; this paper specifies its API surface and its agent build plan, but does not report measured properties of an implementation that does not yet exist. Numbers attributed to `libyaml-safer` are external results we cite; numbers attributed to `safe-libyaml` are explicit *targets* for the in-progress build.

[1] `https://github.com/dtolnay/unsafe-libyaml`

[2] `https://github.com/simonask/libyaml-safer`

1.3 Outline

Section 2 fixes notation (lifetime regions, the borrow judgment, the `Result`/`Option` algebra) as a clarity tool; it is not a new formal system. Section 3 treats ownership and borrowing for streaming parsers; Section 4 dives into the lifetime-parameterised `Event<'a>` enum, the central design decision. Section 5 addresses error handling; Section 6 maps libyaml's STACK/QUEUE/STRING macros to safe `std` types. Section 7 discusses `no_std + alloc`. Section 8 treats tagged unions vs. `enum`. Section 9 formalises the `unsafe` discipline. Section 10 specifies the `#[repr(C)]` ABI shim and `cbindgen` integration. Section 11 surveys the quality-gate tools. Section 12 situates the work alongside `nom`, `winnow`, `rustls`, `rav1d`, and others. Section 13 states the safety results and Section 14 discusses limitations. Section 15 concludes. Appendix A is the mandatory *Development Plan for Prototype Agents* covering the scaffold and idiom phase.

2 Notation: A Lightweight Framework

What this section is and is not. The fragment of formalism introduced below is a *notational tool*: it lets later sections state precisely which parts of the design are forced by Rust's type system and which are engineering choices. We do not introduce any new formal system, nor do we prove any new property of Rust itself. Where Rust's full type system has been formalised — most completely by the RustBelt project [1] — we cite their results rather than re-derive them. The notation is inspired by region-based memory management [3] and is adequate for the API-design statements that follow.

2.1 Lifetimes as regions

Definition 2.1 (Region)**.** A *region* (or *lifetime*) $\rho \in \mathcal{R}$ is an element of a partially-ordered set $(\mathcal{R}, :)$ where $\rho_1 : \rho_2$ (read "ρ_1 outlives ρ_2") means every program point at which a borrow with lifetime ρ_2 is live is dominated by a program point at which a borrow with lifetime ρ_1 is live. The Rust surface syntax writes $'a : 'b$.

Remark 2.2 (Asymmetry of the colon). The Rust syntax overloads " : " between type ascription ($x : \tau$) and lifetime constraints ($'a : 'b$). We write $:$ for the latter throughout to avoid ambiguity.

Definition 2.3 (Static region)**.** The distinguished region $'static \in \mathcal{R}$ satisfies $'static : \rho$ for every $\rho \in \mathcal{R}$. Values of type $\&'^{static}\, T$ are valid for the entire program execution.

2.2 The borrow judgment

Definition 2.4 (Typing context)**.** A typing context Γ is a finite map from variables to pairs (τ, ρ) where τ is a Rust type and ρ is the region during which the binding is live. We write $\Gamma, x : \tau\langle\rho\rangle$ for the extension.

The borrow checker can be presented as a judgment $\Gamma \vdash e : \tau\langle\rho\rangle$, read "in context Γ, expression e has type τ valid in region ρ". Three rules suffice for our purposes. *Notational*

bridge: the formal $\&^{\rho}\, T$ in this section corresponds to Rust's surface syntax `&'a T` (reading ρ as $'a$); $\&\mathsf{mut}\, T$ corresponds to `&'a mut T`. The remainder of the paper uses the Rust surface syntax exclusively; the formal notation appears only in this section and in the proofs of Propositions 2.5, 4.1, and 9.1.

$$\textsc{Var}\ \frac{(x : \tau\langle\rho\rangle) \in \Gamma}{\Gamma \vdash x : \tau\langle\rho\rangle} \qquad \textsc{Borrow}\ \frac{\Gamma \vdash e : \tau\langle\rho\rangle \qquad \rho' : \rho}{\Gamma \vdash \&\, e : \&^{\rho'}\, \tau\langle\rho'\rangle} \qquad \textsc{Sub}\ \frac{\Gamma \vdash e : \tau\langle\rho\rangle \qquad \rho : \rho'}{\Gamma \vdash e : \tau\langle\rho'\rangle}$$

Proposition 2.5 (Variance of borrow types)**.** Shared *borrows are* covariant *in their lifetime parameter: if* $\rho_1 : \rho_2$ *then* $\&^{\rho_1}\, \tau <: \&^{\rho_2}\, \tau$. Mutable *borrows are* invariant *in their lifetime parameter: if* $\&^{\rho_1}\, \mathsf{mut}\, \tau <: \&^{\rho_2}\, \mathsf{mut}\, \tau$ *then* $\rho_1 = \rho_2$. *The invariance of the mutable case arises because* $\&\mathsf{mut}\, \tau$ *supports both reads (which would justify covariance)* and *writes (which, symmetrically, would justify contravariance); the conjunction of covariance and contravariance is invariance.*

Proof sketch. Subtype substitution must preserve typing in every position. For shared borrows, only the read position is exercised; a longer lifetime always admits the uses a shorter one would, hence covariance. For mutable borrows, the same reasoning gives covariance for the read position but contravariance for the write position: a write through a long-lived alias must be visible through every other alias of the same cell, so shrinking the lifetime in a write-reachable position is unsound. The conjunction of the two forces invariance. The full argument is RustBelt's main soundness theorem [1]. □

2.3 The Result and Option algebra

Definition 2.6 (Sum-type error algebra)**.** For Rust types T, E we have the sum types

$$\mathsf{Result}\langle T,\, E\rangle \;=\; \mathsf{Ok}(T) + \mathsf{Err}(E), \qquad \mathsf{Option}\langle T\rangle \;=\; \mathsf{Some}(T) + \mathsf{None}.$$

The `?` operator implements the Kleisli composition $f \,?\, g : \mathsf{Result}\langle T,\, E\rangle \to \mathsf{Result}\langle U,\, E\rangle$, lifting $g : T \to \mathsf{Result}\langle U,\, E\rangle$ across the error monad and short-circuiting on Err.

Proposition 2.7 (Errors compose via `?`)**.** *Let* $f_1 : A \to \mathsf{Result}\langle B,\, E_1\rangle$ *and* $f_2 : B \to \mathsf{Result}\langle C,\, E_2\rangle$. *If the crate's error type* E *satisfies both* $\mathsf{From}\langle E_1\rangle$ *and* $\mathsf{From}\langle E_2\rangle$, *then the composition* $f_2 \circ f_1$ *has a canonical lift to* $A \to \mathsf{Result}\langle C,\, E\rangle$ *via the* `?` *operator alone:*

```
fn compose(a: A) -> Result<C, E> {
    let b = f1(a)?; // ? invokes E::from(E1)
    let c = f2(b)?; // ? invokes E::from(E2)
    Ok(c)
}
```

The `?` *operator desugars to a match that invokes* `E::from(e1)` *(resp.* `E::from(e2)`*) on the* `Err` *branch, giving the uniform short-circuit on the error monad.*

This is the formal justification for the `thiserror` pattern discussed in Section 5: deriving `#[from]` on each nested error arm installs the required `From` instance.

3 Ownership and Borrowing for Streaming Parsers

A streaming parser sits at the intersection of three resource concerns: an *input buffer* that someone must own, an emitted *event stream* that must be either owned or borrowed back into the buffer, and an *internal state* (token queue, indent stack, mark stack) that is exclusively the parser's. The C `libyaml` answer is to make every pointer raw; the safe-Rust answer is to assign each concern to exactly one of the four ownership classes.

Definition 3.1 (Pointer ownership classification)**.** For each C pointer p in a function we assign one of:

- Owning: p is allocated in the function and freed in the function (or returned to a caller who is responsible for freeing it). Rust target: $\mathsf{Box}\langle T\rangle$ or owned $\mathsf{Vec}\langle T\rangle$.
- BorrowShared: p is read but never written through and never freed. Rust target: $\&'a\, T$ or $\&'a\, [T]$.
- BorrowMut: p is written through but not freed. Rust target: $\&\mathsf{mut}\ T$ or $\&\mathsf{mut}\ [T]$.
- SharedOwnership: p is held by multiple owners with potentially overlapping lifetimes; Rust target: $\mathsf{Rc}\langle T\rangle$ or $\mathsf{Arc}\langle T\rangle$.

For `libyaml`'s parser, the classification falls out cleanly: the input buffer is owned by the `Parser`; the token queue and the indent and mark stacks are owned by the `Parser`; the events emitted to the caller are either *borrowed* from the buffer (zero-copy, lifetime-parameterised), or *owned* (allocated string copies). This dichotomy is the central design tension addressed in Section 4.

3.1 The single-owner buffer

The C `yaml_parser_set_input_string(parser, input, length)` accepts a (`ptr`, `len`) pair without specifying ownership. The Rust analogue must choose. We propose three constructors:

```
impl Parser {
    /// Parse from a borrowed byte slice. The Parser borrows from ‘input‘
    /// for the lifetime of the Parser; events may borrow zero-copy.
    pub fn from_slice<’a>(input: &’a [u8]) -> Parser<’a>;

    /// Parse from an owned byte vector. The Parser owns the buffer.
    /// Events that need to borrow scalar text will copy or be ‘OwnedEvent‘.
    pub fn from_vec(input: Vec<u8>) -> Parser<’static>;

    /// Parse from any reader. The Parser maintains its own internal buffer.
    pub fn from_reader<R: std::io::Read + ’static>(r: R) -> Parser<’static>;
}
```

The lifetime parameter on `Parser<'a>` expresses the zero-copy guarantee that an event borrowed from the parser cannot outlive the input. The reader-based constructor degenerates to $'static$ because the parser owns the buffer.

3.2 The zero-copy `&[u8]` pattern

Several Rust parser ecosystems (`nom` [4], `winnow`, `httparse`, `serde_json`'s borrowed mode) standardise on the `&'a [u8]` input pattern: the parser receives a borrowed byte slice and emits structured values that may borrow back into that same slice. This is the natural Rust expression of streaming parsing without allocation: the input lives at lifetime $'a$, every output node that is a substring of the input lives at the same $'a$, and the borrow checker statically prevents the slice from being freed while events that point into it are live.

For YAML, this pattern is partially achievable: most plain scalars are substrings of the input, but escape-processed scalars (e.g. `"line\nbreak"`) require an owned `String` because the decoded form does not appear contiguously in the input. We handle this with the `Cow` pattern in the next section.

4 Lifetimes for Event-Based APIs

4.1 The lifetime-parameterised `Event<'a>`

The central public type of `safe-libyaml` is the parser event enum. Its design is the most consequential decision in the crate; every downstream choice flows from it. We propose:

```
use std::borrow::Cow;

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Event<'a> {
    StreamStart {
        encoding: Encoding,
    },
    StreamEnd,
    DocumentStart {
        version: Option<VersionDirective>,
        tags: Vec<TagDirective<'a>>,
        implicit: bool,
    },
    DocumentEnd {
        implicit: bool,
    },
    Alias {
        anchor: Cow<'a, str>,
    },
    Scalar {
        anchor: Option<Cow<'a, str>>,
        tag: Option<Cow<'a, str>>,
        value: Cow<'a, str>,
        plain_implicit: bool,
        quoted_implicit: bool,
        style: ScalarStyle,
    },
```

```
    SequenceStart {
        anchor: Option<Cow<'a, str>>,
        tag: Option<Cow<'a, str>>,
        implicit: bool,
        style: SequenceStyle,
    },
    SequenceEnd,
    MappingStart {
        anchor: Option<Cow<'a, str>>,
        tag: Option<Cow<'a, str>>,
        implicit: bool,
        style: MappingStyle,
    },
    MappingEnd,
}
```

The `Cow<'a, str>` (clone-on-write) type is the linchpin: it is either `Borrowed(&'a str)` pointing into the input buffer, or `Owned(String)` when the parser had to decode an escape sequence. The borrow checker enforces that a `Borrowed` variant cannot outlive the input.

Proposition 4.1 (Soundness of Cow-events)**.** *Suppose* `Parser<'a>` *borrows input* $b : \&'^{a}\,[u8]$*, and* parse : &mut Parser$\langle 'a\rangle \to$ Result$\langle$Event$\langle 'a\rangle$, Error$\rangle$*. Then for any* e : Event$\langle 'a\rangle$ *produced by* `parse`*, every* `Cow::Borrowed(s)` *sub-component of* e *satisfies* $s : \&'^{a}$ str*, hence the borrow checker prevents* b *from being dropped while any such* e *is live.*

Proof sketch. By the Borrow and Sub rules of Section 2, the lifetime $'a$ on the input flows through the `Parser<'a>` type into the return type of `parse`. The `Cow` type is parameterised covariantly in $'a$, so Cow$\langle 'a\rangle <:$ Cow$\langle 'b\rangle$ whenever $'a : 'b$. The drop-check for b fails if any reference parameterised by $'a$ is live, by the NLL rules. □

4.2 Escape hatches: `OwnedEvent` and the borrow-vs-clone tradeoff

There are circumstances under which holding an event across calls to `parse` is desirable: a wrapper that buffers events for replay, a multi-threaded reader/parser pipeline, or a `serde`-style deserialiser that constructs a tree from the event stream. In these cases the lifetime parameter on `Event<'a>` becomes a usability tax: every helper struct grows the lifetime parameter virally.

The standard escape hatch is to provide a sibling `OwnedEvent` type with all `Cow`s collapsed to `String`, and a single conversion `Event::into_owned`. This pattern is the Rust analogue of the "borrow now, copy later if you want to escape" discipline.

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct OwnedEvent {
    inner: Event<'static>, // Cow::Owned variants only
}

// Helper that lifts a Cow<'a, str> into a Cow<'static, str> by always cloning.
#[inline]
```

```
fn own(c: Cow<'_, str>) -> Cow<'static, str> { Cow::Owned(c.into_owned()) }

impl<'a> Event<'a> {
    pub fn into_owned(self) -> OwnedEvent {
        let inner: Event<'static> = match self {
            Event::StreamStart { encoding } =>
                Event::StreamStart { encoding },
            Event::StreamEnd =>
                Event::StreamEnd,
            Event::DocumentStart { version, tags, implicit } =>
                Event::DocumentStart {
                    version,
                    tags: tags.into_iter()
                        .map(|t| TagDirective { handle: own(t.handle),
                                                prefix: own(t.prefix) })
                        .collect(),
                    implicit,
                },
            Event::DocumentEnd { implicit } =>
                Event::DocumentEnd { implicit },
            Event::Alias { anchor } =>
                Event::Alias { anchor: own(anchor) },
            Event::Scalar { anchor, tag, value,
                            plain_implicit, quoted_implicit, style } =>
                Event::Scalar {
                    anchor: anchor.map(own),
                    tag: tag.map(own),
                    value: own(value),
                    plain_implicit, quoted_implicit, style,
                },
            Event::SequenceStart { anchor, tag, implicit, style } =>
                Event::SequenceStart {
                    anchor: anchor.map(own), tag: tag.map(own),
                    implicit, style,
                },
            Event::SequenceEnd =>
                Event::SequenceEnd,
            Event::MappingStart { anchor, tag, implicit, style } =>
                Event::MappingStart {
                    anchor: anchor.map(own), tag: tag.map(own),
                    implicit, style,
                },
            Event::MappingEnd =>
                Event::MappingEnd,
        };
        OwnedEvent { inner }
    }
```

```
}
```

Remark 4.2 (The `libyaml-safer` insight). The breakthrough of S. Ask's `libyaml-safer` port was to recognise that the C `yaml_emitter_t` struct contains an internal analysis record (`yaml_scalar_analysis_t`) holding raw `char*` pointers back into the event being emitted. The naive Rust transcription is self-referential and impossible without `Pin`. The solution is to factor analysis into an `Analysis<'a>` struct that borrows from the `Event<'a>` and is passed explicitly through emitter functions. The Rust borrow checker forces this re-architecture, which is in fact a strictly better design.

```
struct Analysis<'a> {
    anchor: Option<&'a str>,
    tag: Option<&'a str>,
    value: &'a str,
    style: ScalarStyle,
    multiline: bool,
    flow_plain_allowed: bool,
    block_plain_allowed: bool,
    single_quoted_allowed: bool,
    block_allowed: bool,
}

impl Emitter {
    fn analyze<'a>(&self, ev: &'a Event<'a>) -> Result<Analysis<'a>, Error>;

    fn emit_scalar<'a>(
        &mut self,
        ev: &'a Event<'a>,
        analysis: &Analysis<'a>,
    ) -> Result<(), Error>;
}
```

The signature of `emit_scalar` now *documents* that `analysis` borrows from the same event `ev`; this is invisible in the C code and is a typical example of how the borrow checker promotes implicit aliasing to compile-time-checked structure.

5 Error Handling: thiserror vs. anyhow

5.1 Library vs. application errors

The Rust ecosystem has converged on a clean dichotomy:

- `thiserror` for *library* crates that define their own `Error` type. Callers can pattern-match on variants. Errors are *enumerable, structured, semver-stable*.
- `anyhow` for *application* code that just needs to propagate errors with context. Errors are *opaque, dynamic, ergonomic*.

`safe-libyaml` is a library and so must use `thiserror`. The error type is the public face of the crate; downstream users (`serde_yaml_ng` etc.) will pattern-match on its variants to distinguish reader errors from scanner errors from parser errors.

```
use thiserror::Error;

#[derive(Debug, Error)]
pub enum Error {
    #[error("reader error at line {line}, column {column}: {problem}")]
    Reader { problem: String, line: u64, column: u64 },

    #[error("scanner error at line {line}, column {column}: {problem}")]
    Scanner { problem: String, line: u64, column: u64,
              context: Option<String>, context_mark: Option<Mark> },

    #[error("parser error at line {line}, column {column}: {problem}")]
    Parser { problem: String, line: u64, column: u64,
              context: Option<String>, context_mark: Option<Mark> },

    #[error("composer error at line {line}, column {column}: {problem}")]
    Composer { problem: String, line: u64, column: u64 },

    #[error("emitter error: {problem}")]
    Emitter { problem: String },

    #[error("io error: {0}")]
    Io(#[from] std::io::Error),

    #[error("memory error: allocation of {0} bytes failed")]
    Memory(usize),
}
```

5.2 Why we do *not* chain via `Error::source()`

A common Rust style is to chain errors via the `std::error::Error::source()` method. For `safe-libyaml` we deliberately flatten the error structure: the C `yaml_parser_t` stores a single problem with an optional "context" problem alongside, and the variant fields above mirror that shape exactly. This (a) preserves byte-for-byte error semantics with libyaml, (b) avoids dynamic dispatch through `dyn Error`, and (c) makes `Error: Send + Sync + 'static` trivially.

5.3 The `Result<Event<'a>, Error>` shape

The natural signature for the parser entry point is:

```
impl<'a> Parser<'a> {
    #[must_use = "events that are not consumed are silently dropped"]
```

```
    pub fn parse(&mut self) -> Result<Event<'a>, Error>;
}
```

The `#[must_use]` attribute is essential: it converts the C-style "ignored return value" bug class into a compile-time warning. The lifetime $'a$ on the returned event is inherited from the parser, which in turn was set at construction.

Example (Driving the parser). A typical consumer loop:

```
let input = std::fs::read("config.yaml")?;
let mut p = safe_libyaml::Parser::from_slice(&input);
loop {
    match p.parse()? {
        Event::StreamEnd => break,
        Event::Scalar { value, .. } => println!("{value}"),
        _ => {}
    }
}
```

The `?` operator propagates I/O errors and parse errors uniformly.

6 Memory Management: From C Macros to `std`

6.1 The `libyaml` macro family

The C source defines a family of capacity-doubling collection macros in `yaml_private.h`:

- `STRING_INIT`, `STRING_EXTEND`, `STRING_DEL`, `STRING_JOIN`: byte buffers (start/end/pointer triples).
- `STACK_INIT`, `STACK_PUSH`, `STACK_POP`, `STACK_DEL`: initial capacity 16.
- `QUEUE_INIT`, `ENQUEUE`, `DEQUEUE`, `QUEUE_DEL`: initial capacity 16.

Each macro hand-rolls a typed dynamic array with manual realloc. The naive `c2rust` translation preserves the raw pointer arithmetic verbatim; `unsafe-libyaml` contains hundreds of these expansions. The safe-Rust translation is mechanical:

C macro	Rust replacement	Safety property
`yaml_string_t`	`Vec<u8>`	length-checked indexing
`STRING_INIT(s, n)`	`Vec::with_capacity(n)`	no UAF
`STRING_EXTEND(s)`	`s.reserve(1)`	no double-free
`STRING_DEL(s)`	*drop on scope exit*	no leak
`yaml_stack_t`	`Vec<T>`	length-checked indexing
`STACK_INIT(s, T)`	`Vec::<T>::with_capacity(16)`	–
`STACK_PUSH(s, v)`	`s.push(v)`	–
`STACK_POP(s)`	`s.pop().ok_or(Error::...)?`	no panic on empty
`yaml_queue_t`	`VecDeque<T>`	length-checked indexing
`QUEUE_INIT(q, T)`	`VecDeque::<T>::with_capacity(16)`	–
`ENQUEUE(q, v)`	`q.push_back(v)`	–
`DEQUEUE(q)`	`q.pop_front().ok_or(Error::...)?`	no panic on empty

We deliberately avoid `.unwrap()` as the canonical translation for `STACK_POP` and `DEQUEUE`. `Vec::pop` returns an `Option<T>`; the safe-Rust idiom uses `.ok_or(...)?` to convert "empty stack" (a parser-state-machine inconsistency) into a typed `Error` variant rather than a panic. Reserving `.unwrap()` for genuine "cannot happen" invariants is part of the project's lint policy (see Appendix A's `lints.toml`), and panics in a streaming parser would be a denial-of-service vector against any caller that feeds adversarial input.

6.2 Capacity-doubling proposition

Proposition 6.1 (Amortised constant push)**.** *For `Vec::push` with the standard doubling growth policy, the amortised cost of n consecutive pushes is $O(n)$ total, hence $O(1)$ per push. This matches the asymptotic behaviour of `libyaml`'s `STACK_PUSH` macro and is a textbook result; we restate it only to make explicit that the `Vec` substitution incurs no asymptotic performance cost.*

Proof. Standard accounting argument: pre-charge each push with 3 tokens; one pays for the push itself, two are saved. When the array doubles from capacity k to $2k$, we have $2k$ saved tokens (two per past push), exactly enough to copy k elements. Hence total cost over n pushes is at most $3n$. □

6.3 When `Cow` earns its keep

We deploy `Cow<'a, str>` (rather than always-`String`) specifically in the public `Event<'a>` fields where zero-copy is achievable for the common case (plain unquoted scalars are substrings of the input) and fallback-to-owned is required for the uncommon case (escape-processed scalars). Internally the parser uses raw `Vec<u8>`/`String` buffers; the `Cow` appears only at the public API boundary.

6.4 The `BString` alternative and the UTF-8 default

For YAML, scalar *values* are not unconditionally guaranteed to be UTF-8: the YAML 1.1 spec admits the YAML stream encoded in UTF-8, UTF-16, or UTF-32, and within double-

quoted scalars permits escape-encoded byte sequences whose decoded form is not necessarily a valid UTF-8 string. Two reasonable designs are possible:

1. *Strict UTF-8 default*: scalar values are `Cow<'a, str>`; non-UTF-8 input is a parse error. This matches `libyaml-safer` and `serde_yaml`'s historical behaviour.

2. *Lenient bytes default*: scalar values are `Cow<'a, [u8]>` or the `bstr::BStr` type. Non-UTF-8 input is preserved verbatim.

The choice and its justification. `safe-libyaml` adopts (1) as the *default* type, and exposes a sibling `ByteEvent<'a>` type (with `Cow<'a, [u8]>` payloads) for the lenient case, with a `TryFrom<ByteEvent<'a»` impl that produces an `Event<'a>` on UTF-8-validatable input. Three engineering reasons make (1) the right default:

1. *Drop-in compatibility with the consumer ecosystem and the `serde` data model.* The downstream forks (`serde_yaml_ng`, `serde_yaml_bw`, `serde_norway`) already produce `String`-typed outputs in their public APIs. More fundamentally, `serde`'s data model treats string-typed values as valid UTF-8 by construction (via `Visitor::visit_str(&str)`); a bytes-first parser would force every consumer to insert UTF-8 validation at the boundary *anyway*, defeating the purpose of a drop-in replacement and adding the cost twice.

2. *Empirical input distribution.* The Kubernetes, Ansible, and CI/CD-pipeline use cases that generate the bulk of real YAML traffic emit UTF-8 by construction: they originate from JSON-like serialisers, from keyboard-typed configuration, or from build-system templates. Non-UTF-8 double-quoted scalars are exceedingly rare in practice; pessimising the common case to accommodate them inverts the cost ratio.

3. *Failure mode is recoverable, not silent.* If a consumer encounters non-UTF-8 input under the default API, they receive a typed `Error::Scanner` variant pointing to the exact offset, and may either reject the input (often the right answer) or re-parse with the `ByteEvent` API. The failure surface is observable and auditable, in contrast to the C library's silent acceptance of invalid UTF-8 that may corrupt downstream consumers expecting valid strings.

Impact on legacy YAML. Files that intentionally embed non-UTF-8 byte sequences in scalars — a niche but real use case for binary blobs encoded in YAML for transport — must use the `ByteEvent` API explicitly, or use the standard YAML 1.2 `!!binary` tag with base64-encoded ASCII payload (which round-trips through the strict-UTF-8 API trivially). We document this trade-off prominently in the crate's README.

Why not lenient by default with strict opt-in? A lenient default produces ergonomic friction at the consumer side: every downstream function that expects `&str` would need a `from_utf8` call. Inverting the default trades a small loss in spec coverage for a large gain in caller ergonomics; for a library whose 90M-download ecosystem is built around `serde`-style `String` fields, this is the correct trade.

7 no_std + alloc

7.1 What changes when shipping for embedded

Many YAML use cases are embedded: configuration on small Linux SBCs, robot control fleets, IoT firmware. We make `safe-libyaml` compatible with `#![no_std]` provided the target supports an allocator (`alloc` crate). Pure no-allocation YAML parsing is a separate research problem; we do not attempt it here.

```
#![cfg_attr(not(feature = "std"), no_std)]
extern crate alloc;
use alloc::{vec::Vec, collections::VecDeque, string::String, borrow::Cow};
use core::fmt;
```

7.2 The `core::ffi` story

The `core::ffi` module (stabilised in Rust 1.64) provides `c_char`, `c_int`, `c_void`, etc. without requiring `std`. This means the FFI shim crate `safe-libyaml-sys` can also be `no_std`-compatible, important for embedded users who want a drop-in C ABI.

7.3 Feature flags

The `Cargo.toml` declares a clean feature matrix:

```
[features]
default = ["std"]
std = [] # opt out for no_std targets
serde = ["dep:serde"] # serde::Serialize/Deserialize for Event
ffi = [] # build the C ABI exports
fuzzing = [] # extra invariant checks for fuzz harness
```

Remark 7.1 (Avoid the trait-object `Box<dyn Read>` trap). The natural Rust signature for "parse from any reader" is `Box<dyn Read + 'static>`, which requires heap allocation and dynamic dispatch. This is fine for std targets but nontrivial in `no_std + alloc`. We provide `Parser::from_reader<R: Read>` generic over `R` (monomorphised) as the no_std-friendly alternative.

8 Tagged Enums vs. C Unions

8.1 The discriminant

A C tagged union is two related declarations:

```
typedef struct {
    yaml_event_type_t type;
    union {
```

```
        struct { /* stream_start data */ } stream_start;
        struct { /* document_start data */ } document_start;
        ...
        struct { /* scalar data */ } scalar;
    } data;
    yaml_mark_t start_mark, end_mark;
} yaml_event_t;
```

The `type` field is the discriminant, but the C compiler does not enforce that callers read only the union variant matching the discriminant. Reading the wrong variant is undefined behaviour. The Rust translation *fuses* the discriminant and the data:

```
pub enum Event<'a> {
    StreamStart { encoding: Encoding },
    StreamEnd,
    DocumentStart { /* ... */ },
    DocumentEnd { implicit: bool },
    Alias { anchor: Cow<'a, str> },
    Scalar { /* ... */ },
    SequenceStart { /* ... */ },
    SequenceEnd,
    MappingStart { /* ... */ },
    MappingEnd,
}
```

Proposition 8.1 (Match exhaustiveness as a safety multiplier)**.** *Suppose E is a Rust enum with n variants and a* `match` *expression* `m` *on a value of type E. The Rust compiler statically rejects* `m` *unless every variant is reachable from at least one arm (potentially via wildcard* `_`*). Adding a new variant to E in a non-breaking-major version requires the* `#[non_exhaustive]` *attribute; otherwise downstream* `match` *expressions break and the compiler points to every site requiring an update. This is a standard property of Rust's pattern-matching design [1]; we restate it because it is the reason a tagged-union → enum translation upgrades silent C bugs into compile-time errors.*

This is the key safety multiplier: when libyaml adds a hypothetical new event (say, `YAML_PARSE_DIRECTIVE_EVENT`), every C consumer silently breaks; the Rust crate's downstream consumers instead get a compile error pointing to the exact site needing a new arm. The cost of API evolution is paid once at compile time, not many times at runtime.

8.2 The `#[non_exhaustive]` discipline

For the public `Event` enum we recommend `#[non_exhaustive]` to permit non-breaking variant addition; for the internal `Token` enum (not part of the public API) we omit the attribute so the compiler can exhaustively verify scanner → parser dataflow.

```
#[non_exhaustive]
#[derive(Debug, Clone, PartialEq, Eq)]
```

```rust
pub enum Event<'a> { /* variants as above */ }

// Internal -- exhaustiveness checked by compiler:
#[derive(Debug, Clone, PartialEq, Eq)]
pub(crate) enum Token<'a> { /* 22 token kinds */ }
```

9 Unsafe Boundaries

9.1 Discipline

The crate adopts the following lint configuration in the root module:

```rust
#![deny(unsafe_code)] // no unsafe in the safe crate
#![deny(missing_docs)]
#![deny(rust_2018_idioms)]
#![warn(clippy::pedantic, clippy::nursery)]
```

The single `#![deny(unsafe_code)]` line is the most consequential: it instructs the compiler to reject every `unsafe` block in the crate, full stop. Any time the implementation needs unsafe, the developer must justify lifting the deny — a workflow that surfaces every unsafe introduction to code review.

9.2 The `safe-libyaml-sys` split

The C ABI exports must be `unsafe` (the safety contract is on the caller, who must maintain raw-pointer validity). We therefore split the workspace:

- `safe-libyaml` — the safe Rust crate; `#![deny(unsafe_code)]`. Internal use only of safe `std` types.
- `safe-libyaml-sys` — the FFI shim; `#![deny(unsafe_op_in_unsafe_fn)]` forces every dereference inside an `extern "C"` function to live inside an explicit `unsafe { ... }` block with a `// SAFETY:` comment explaining the invariant. Re-exports nothing of substance; merely wraps `safe-libyaml`'s safe API in C symbols.

Proposition 9.1 (Compositional safety)**.** *Suppose every* ***`unsafe`*** *block in* ***`safe-libyaml-sys`*** *satisfies its documented* ***`// SAFETY:`*** *invariants, and* ***`safe-libyaml`*** *contains no* ***`unsafe`*** *blocks. Then any data race, use-after-free, or out-of-bounds access in a Rust caller of* ***`safe-libyaml`*** *is impossible unless caused by a* C *caller violating the FFI contract. In particular, pure-Rust callers cannot trigger any UB inherited from* ***`libyaml`****'s C behaviour.*

Proof sketch. Rust's type soundness theorem [1] states that well-typed safe code cannot exhibit UB. `safe-libyaml` is well-typed safe code by construction (`#![deny(unsafe_code)]`). The composition of safe Rust and safe Rust is safe Rust. The only entry points for UB are the `unsafe extern "C"` functions in the sys crate, whose preconditions are enforced on the caller. □

9.3 Interior unsafety as an anti-pattern

A common `c2rust` output uses `unsafe { }` blocks deep inside the parser to wrap raw pointer arithmetic. This is the *wrong* boundary: a panic or misuse propagates back through allegedly safe wrappers. The `safe-libyaml` discipline is to prohibit this entirely. If ergonomic considerations seem to demand interior unsafety, the right response is to refactor the data structure (e.g. replace a self-referential struct with a parameter-passed `Analysis<'a>` per Remark 4.2).

10 ABI-Compatible C Export

10.1 The drop-in replacement principle

`libyaml`'s public header `yaml.h` defines roughly 80 `extern` symbols: parser/emitter constructors and destructors, event constructors, document tree manipulators, and version-info accessors. A *drop-in* Rust replacement is a crate that, when compiled with the `ffi` feature, produces a `libsafe_yaml.so` (or `.dylib` or `.dll`) with byte-identical exported symbols. C callers and existing Rust crates such as `serde_yaml_ng` (which links against `unsafe-libyaml`) can be relinked against `safe-libyaml-sys` without modification.

10.2 `#[repr(C)]` layout

```
#[repr(C)]
pub struct yaml_parser_t {
    inner: *mut crate::Parser<'static>, // opaque to C
    error: yaml_error_type_t,
    problem: *const c_char,
    problem_offset: size_t,
    problem_value: c_int,
    problem_mark: yaml_mark_t,
    context: *const c_char,
    context_mark: yaml_mark_t,
    /* padding to match libyaml struct size */
    _padding: [u8; PARSER_PADDING],
}

#[repr(C)]
pub struct yaml_mark_t {
    pub index: size_t,
    pub line: size_t,
    pub column: size_t,
}

#[repr(C, u32)]
pub enum yaml_event_type_t {
    YAML_NO_EVENT = 0,
    YAML_STREAM_START_EVENT,
```

```
    YAML_STREAM_END_EVENT,
    YAML_DOCUMENT_START_EVENT,
    YAML_DOCUMENT_END_EVENT,
    YAML_ALIAS_EVENT,
    YAML_SCALAR_EVENT,
    YAML_SEQUENCE_START_EVENT,
    YAML_SEQUENCE_END_EVENT,
    YAML_MAPPING_START_EVENT,
    YAML_MAPPING_END_EVENT,
}
```

10.3 The `cbindgen` contract

We commit a `cbindgen.toml` to the sys crate:

```
language = "C"
header = "/* SPDX-License-Identifier: MIT -- safe-libyaml */"
include_guard = "SAFE_YAML_H_INCLUDED"
pragma_once = true
cpp_compat = true
style = "tag"

[export]
prefix = "yaml_" # match libyaml's symbol prefix
include = ["yaml_parser_t", "yaml_emitter_t", "yaml_event_t",
           "yaml_document_t", "yaml_node_t", "yaml_mark_t",
           "yaml_error_type_t", "yaml_event_type_t",
           "yaml_scalar_style_t", "yaml_sequence_style_t",
           "yaml_mapping_style_t", "yaml_encoding_t"]

[parse]
parse_deps = false
```

A CI job runs:

```
cbindgen --config cbindgen.toml --crate safe-libyaml-sys \
        --output target/yaml.h
diff -u reference/libyaml/include/yaml.h target/yaml.h \
       > target/yaml-h-diff.txt
```

and fails the build if the diff exceeds an allow-list of cosmetic differences (whitespace, comments, ordering). This is the formal *ABI parity gate*.

10.4 Symbol-level enumeration

```
nm -D libsafe_yaml.so | awk '$2=="T"{print $3}' | sort > target/syms.rust
nm -D libyaml.so.0 | awk '$2=="T"{print $3}' | sort > target/syms.c
```

```
diff -u target/syms.c target/syms.rust
```

must produce empty output for a true drop-in. The CI gate is identical to the cbindgen check.

10.5 Bridging C `FILE*` to Rust `Read`

`libyaml`'s public API includes `yaml_parser_set_input_file(parser, FILE *file)`, and any ABI-faithful drop-in must export the same signature. The challenge is that `FILE*` is an opaque C-stdlib handle whose internals are not specified; we cannot read it from safe Rust directly. The `-sys` crate solves this by wrapping the `FILE*` in a thin `std::io::Read` adapter:

```
struct CFileReader { file: *mut libc::FILE }

impl std::io::Read for CFileReader {
    fn read(&mut self, buf: &mut [u8]) -> std::io::Result<usize> {
        // SAFETY: `self.file` is a non-null FILE* whose validity is
        // the responsibility of the C caller (precondition of
        // yaml_parser_set_input_file). `fread` is bounded by buf.len().
        let n = unsafe {
            libc::fread(buf.as_mut_ptr().cast(), 1, buf.len(), self.file)
        };
        // SAFETY: `ferror` only inspects the FILE* and returns int.
        if n == 0 && unsafe { libc::ferror(self.file) } != 0 {
            return Err(std::io::Error::last_os_error());
        }
        Ok(n)
    }
}

#[no_mangle]
pub unsafe extern "C" fn yaml_parser_set_input_file(
    parser: *mut yaml_parser_t,
    file: *mut libc::FILE,
) {
    // SAFETY: `parser` is a valid initialised yaml_parser_t (caller
    // precondition); `file` is a non-null FILE* opened for reading.
    let parser = unsafe { &mut *parser };
    parser.set_reader(Box::new(CFileReader { file }));
}
```

The safety invariants the `unsafe` block must uphold are: (i) `file` is a non-null `FILE*` that remains open and valid for the lifetime of the parser (caller's responsibility, exactly as in the C original); (ii) `buf.as_mut_ptr()` is valid for writes of `buf.len()` bytes (Rust's slice invariants guarantee this); (iii) the `FILE*` is not concurrently accessed from another thread (matching C's stdio thread-safety expectations and libyaml's existing single-threaded

contract).

This adapter pattern keeps the unsafe code-surface tiny — two `libc::fread/ferror` calls — while letting the entire parser body remain in safe Rust. The same pattern in reverse provides `yaml_emitter_set_output_file` via a `CFileWriter: std::io::Write`.

10.6 Why `serde_yaml_ng` matters

The two main forks (`serde_yaml_ng` and `serde_yaml_bw`) together account for roughly 90M downloads of YAML parsing in the Rust ecosystem; both currently depend on `unsafe-libyaml`. If `safe-libyaml-sys` is a perfect drop-in, the change for these crates is a single line in their `Cargo.toml`:

```
- unsafe-libyaml = "0.2"
+ safe-libyaml-sys = "0.1"
```

This is the engineering target that justifies the effort: a single PR upgrades the security posture of every Rust crate using YAML.

11 Tooling: clippy, rustfmt, cargo-deny, cargo-audit, cargo-semver-checks

`cargo fmt -all - -check`
enforces formatting parity. CI gate.

`cargo clippy - -W clippy::pedantic -W clippy::nursery -D warnings`
surfaces idiom violations. Project allow-list is documented in `lints.toml`.

`cargo deny check`
verifies no banned licenses, no advisories on dependencies, no duplicate dependency versions.

`cargo audit`
checks the RustSec advisory database. Targets zero vulnerabilities.

`cargo semver-checks`
verifies that minor releases do not break the public API. Run pre-release.

`cargo +nightly miri test`
runs the test suite under Miri's Stacked Borrows checker. Required to pass with zero UB.

`cargo fuzz run fuzz_parser - -max_total_time=3600`
CI job runs differential fuzzing against C `libyaml` for an hour.

`cargo bench`
criterion benchmarks gate performance regression at ±5% relative to baseline.

A reference `lints.toml` is provided in Appendix A.

12 Related Work

Parser combinators. `nom` [4] and its successor `winnow` [5] are the dominant parser-combinator libraries. They popularised the `&'a [u8]` input pattern that we adopt; they are not directly applicable to YAML because of the latter's indentation-sensitive grammar but they prove the lifetime-parameterised idiom scales.

`serde` and friends. `serde_json` [6] demonstrates that a streaming parser with both borrowed and owned modes (`Cow<'a, str>`) is feasible and ergonomic. The three maintained forks of the deprecated `serde_yaml` — `serde_yaml_ng`, `serde_yaml_bw`, and `serde_norway` — all target `unsafe-libyaml` as their parser backend, and any of them is a candidate for the first PR.

Pure-Rust YAML parsers. `saphyr-parser` and the higher-level `serde-saphyr` provide a fully-Rust alternative; they do not target ABI compatibility with `libyaml`. Our crate is complementary: it gives the existing C and `serde_yaml_*` ecosystems a zero-friction migration path, where `saphyr` requires API rewrites.

Memory-safety initiatives. The Prossimo / ISRG portfolio (`rustls`, `sudo-rs`, `ntpd-rs`, `hickory-dns`, `rav1d`, `zlib-rs`, `libbzip2-rs`, `libzstd-rs`) is the relevant prior art for production-grade C-to-Rust replacements. `rav1d` in particular is a 50K-LOC AV1 decoder whose performance matches the C reference `dav1d`; its lifetime discipline informs ours.

Static analysis for ownership inference. Crown [8] (CAV 2023), Laertes [9] (OOPSLA 2023), and the academic line through SACTOR (March 2025), Rustine (2025), EvoC2Rust (2025), and ORBIT (April 2026) inform the upstream analysis that produces our scaffold.

Verified Rust formalisation. RustBelt [1] (POPL 2018) gives the soundness theorem we cite in Proposition 9.1; Miri [2] (POPL 2026) provides the dynamic UB checker; Kani [10] provides bounded model checking for high-assurance predicates.

13 Results

We summarise the design contributions.

Proposition 13.1 (Design summary: properties of the proposed API)**.** *The public API of `safe-libyaml` as specified in Sections 3–10 is* designed *to satisfy the following properties; the propositions and definitions cited inside the list are the design's structural justifications, not empirical verifications of the in-progress build:*

1. Memory safety*: every operation is statically verified by the Rust borrow checker; no `unsafe` block is required in the safe crate (Proposition 9.1).*
2. Zero-copy in the common case*: plain scalars borrow directly from the input buffer (Proposition 4.1).*
3. Exhaustive event handling*: the `Event<'a>` enum forces every match site to handle every variant (Proposition 8.1).*
4. Composable errors*: `Result<Event<'a>, Error>` allows uniform `?`-propagation (Proposition 2.7).*

5. ABI parity*: `safe-libyaml-sys` exports the same C symbols as `libyaml.so` as verified by `cbindgen` diff and `nm` enumeration.*

6. Amortised performance*: collection operations have the same asymptotic complexity as the C macros (Proposition 6.1).*

Corollary 13.2 (Drop-in replacement, conditional)**.** Provided *the implementation realised by Appendix A's plan satisfies the* ABI parity *clause of Proposition 13.1 (verified by `cbindgen` diff and `nm` enumeration in CI), the crates `serde_yaml_ng` and `serde_yaml_bw` can replace their dependency `unsafe-libyaml = "0.2"` with `safe-libyaml-sys = "0.1"` via a single-line patch with no semantic behaviour change observable to their downstream consumers.*

14 Discussion

14.1 Performance: targets, not measurements

We make no measurement claim in this paper. The closest established data point is S. Ask's manual port `libyaml-safer`, which has been informally reported as "largely on par" with `unsafe-libyaml` [7]; this is an external result about a different (manually-written) crate and we cite it only as plausible evidence that the API shape we propose is not inherently slower than the C original. For the in-progress `safe-libyaml` itself, we set explicit *engineering targets* that the verification agents are responsible for measuring once the build exists: within 2× of the C reference is PASS; within 1.5× is GOOD; matching or exceeding C is EXCELLENT (plausible via LLVM's better inlining of monomorphised generics, but not promised). Whether these targets are actually met is an empirical question outside the scope of this design paper; the benchmarker described in Appendix A's Phase C must report the numbers before any performance claim about `safe-libyaml` is warranted.

14.2 Limitations

Several design choices are tensioned:

- The `Cow<'a, str>` shape requires UTF-8; YAML 1.1 permits non-UTF-8 in double-quoted scalars. The `ByteEvent` sibling is a pragmatic workaround but doubles the API surface.
- The lifetime parameter on `Event<'a>` is viral: any struct holding events must also be parameterised. The `OwnedEvent` escape hatch addresses this at the cost of allocation.
- The `#[non_exhaustive]` attribute on `Event` prevents downstream exhaustive matching; this is the explicit price of API evolution.
- `no_std` support requires `alloc`; we do not target truly heap-free YAML parsing (which would need a static-bounded buffer variant of the parser).

14.3 Place in the broader project (optional reading)

This paper is self-contained and stands as a design specification on its own. For readers interested in the broader *Ferrous Bridge* context: a companion paper [12] treats the source side (which C idioms must be recovered), another [13] treats the translation process itself, and a strategic overview [14] positions the work commercially. None of those are prerequisites for the design statements made here; the present paper is intentionally legible to a Rust engineer with no familiarity with the larger pipeline.

15 Conclusion

We have specified the API surface of `safe-libyaml`, a planned automated safe-Rust replacement for `unsafe-libyaml`, in enough detail that the prototype agents listed in Appendix A can begin implementation immediately. The central design moves are:

1. A lifetime-parameterised `Event<'a>` with `Cow<'a, str>` payloads for zero-copy scalars.
2. A `thiserror`-derived `Error` enum mirroring `libyaml`'s problem categories.
3. A `Vec`/`VecDeque` mapping for the C `STACK_*`/`QUEUE_*` macros.
4. A two-crate workspace splitting the safe API (`safe-libyaml`, gated by a crate-level deny of `unsafe_code`) from the FFI shim (`safe-libyaml-sys`) that preserves `libyaml`'s exact C ABI.
5. A CI gate of `rustc` + Miri + clippy + cargo-deny + `cbindgen` diff + a one-hour `cargo-fuzz` run, enforcing every property automatically.

The work composes naturally with related designs that the reader may choose to consult: the source-side companion [12] supplies the ownership classification that the Appendix A OSG-to-Rust emitter consumes; the process-side companion [13] drives a per-function TRANSLATE-TEST-FIX loop against the API surface specified here. Neither is required to read or apply the present paper.

References

[1] R. Jung, J.-H. Jourdan, R. Krebbers, D. Dreyer. *RustBelt: Securing the Foundations of the Rust Programming Language.* POPL 2018.

[2] R. Jung, H. H. Dang, et al. *Miri: Practical Undefined Behavior Detection for Rust.* POPL 2026.

[3] M. Tofte, J.-P. Talpin. *Region-based Memory Management.* Information and Computation 132(2), 1997.

[4] G. Couprie. *nom: A Byte-Oriented, Streaming, Zero-Copy Parser Combinator Library in Rust.* 2014–2026. `https://github.com/rust-bakery/nom`

[5] *winnow: A parser-combinator framework forked from nom.* https://github.com/winnow-rs/winnow

[6] D. Tolnay. *serde_json: A JSON serialization file format.* https://github.com/serde-rs/json

[7] S. Ask. *Notes on porting libyaml to safe Rust.* https://simonask.github.io/libyaml-safer/, February 2024.

[8] H. Zhang, K. Chen, et al. *Ownership Guided C to Rust Translation.* CAV 2023.

[9] M. Emre, R. Schroeder, K. Dewey, B. Hardekopf. *Translating C to Safer Rust.* OOPSLA 2023.

[10] A. Pochhammer, et al. *The Kani Rust Verifier.* Amazon AWS, 2022–2026. https://github.com/model-checking/kani

[11] Immunant, Galois. *c2rust: a C to Rust transpiler.* https://github.com/immunant/c2rust

[12] M. Long. *C Language Semantics for Safe-Rust Translation.* GrokRxiv:2026.04.c [cs.PL], April 2026.

[13] M. Long. *Automated C-to-Rust Transpiling: Pipeline, Patterns, and Verification.* GrokRxiv:2026.04.c-rust-transpiling [cs.PL], April 2026.

[14] M. Long. *Compile-Time Supremacy: A Strategic Architecture for AI-Assisted C→Rust Migration at Scale.* GrokRxiv:2026.04.c2rust-compile-time-supremacy [cs.SE], April 2026.

A Development Plan for Prototype Agents (Scaffold and Idiom Phase)

Status of this appendix. This appendix is *supplementary material*, not a research result. Its purpose is to provide concrete evidence that the design specified in the body of the paper is *implementable* by the kind of agent pipeline we propose: by listing the actual tasks an A2 (Scaffold Builder) agent, an OSG-to-Rust emitter agent, and a C4 (Idiom Polisher) agent would execute, in the format mandated by the project's knowledge-base addendum, we make the design falsifiable in the engineering sense (a reader could fork ferrous-bridge today and check whether the listed tasks compose into the specified API surface). The appendix does not replace experimental validation; it documents the bridge from specification to build.

A.1 The exact public API the agents must target

The agents output a workspace with two crates. We restate the exact public signatures so the agents have an unambiguous target.

A.1.1 `safe-libyaml::Parser`

```
pub struct Parser<'a> {
    /* private fields */
    _marker: core::marker::PhantomData<&'a [u8]>,
}

impl<'a> Parser<'a> {
    pub fn from_slice(input: &'a [u8]) -> Parser<'a>;
    pub fn from_vec(input: Vec<u8>) -> Parser<'static>;
    pub fn from_reader<R: std::io::Read + 'static>(r: R) -> Parser<'static>;

    pub fn set_encoding(&mut self, encoding: Encoding);
    pub fn encoding(&self) -> Encoding;

    #[must_use]
    pub fn parse(&mut self) -> Result<Event<'a>, Error>;

    pub fn mark(&self) -> Mark;
}
```

A.1.2 `safe-libyaml::Event`

```
#[non_exhaustive]
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Event<'a> {
    StreamStart { encoding: Encoding },
```

```
    StreamEnd,
    DocumentStart { version: Option<VersionDirective>,
                    tags: Vec<TagDirective<'a>>,
                    implicit: bool },
    DocumentEnd { implicit: bool },
    Alias { anchor: Cow<'a, str> },
    Scalar { anchor: Option<Cow<'a, str>>,
                    tag: Option<Cow<'a, str>>,
                    value: Cow<'a, str>,
                    plain_implicit: bool,
                    quoted_implicit: bool,
                    style: ScalarStyle },
    SequenceStart { anchor: Option<Cow<'a, str>>,
                    tag: Option<Cow<'a, str>>,
                    implicit: bool, style: SequenceStyle },
    SequenceEnd,
    MappingStart { anchor: Option<Cow<'a, str>>,
                    tag: Option<Cow<'a, str>>,
                    implicit: bool, style: MappingStyle },
    MappingEnd,
}

impl<'a> Event<'a> {
    pub fn into_owned(self) -> OwnedEvent;
    pub fn start_mark(&self) -> Mark;
    pub fn end_mark(&self) -> Mark;
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub struct OwnedEvent { /* Event<'static> with all Cow::Owned */ }
```

A.1.3 safe-libyaml::Emitter

```
pub struct Emitter<W: std::io::Write> { /* private */ }

impl<W: std::io::Write> Emitter<W> {
    pub fn new(out: W) -> Emitter<W>;
    pub fn set_encoding(&mut self, e: Encoding);
    pub fn set_canonical(&mut self, b: bool);
    pub fn set_indent(&mut self, n: u8);
    pub fn set_width(&mut self, n: u32);
    pub fn set_unicode(&mut self, b: bool);
    pub fn set_break(&mut self, b: LineBreak);

    pub fn emit(&mut self, event: Event<'_>) -> Result<(), Error>;
    pub fn flush(&mut self) -> Result<(), Error>;
```

```
    pub fn into_inner(self) -> Result<W, Error>;
}
```

A.1.4 `safe-libyaml-sys::extern "C"` symbols

```
#[no_mangle] pub unsafe extern "C" fn yaml_parser_initialize(
    parser: *mut yaml_parser_t) -> c_int;

#[no_mangle] pub unsafe extern "C" fn yaml_parser_delete(
    parser: *mut yaml_parser_t);

#[no_mangle] pub unsafe extern "C" fn yaml_parser_set_input_string(
    parser: *mut yaml_parser_t,
    input: *const c_uchar, size: size_t);

#[no_mangle] pub unsafe extern "C" fn yaml_parser_set_input_file(
    parser: *mut yaml_parser_t, file: *mut FILE);

#[no_mangle] pub unsafe extern "C" fn yaml_parser_set_encoding(
    parser: *mut yaml_parser_t, encoding: yaml_encoding_t);

#[no_mangle] pub unsafe extern "C" fn yaml_parser_parse(
    parser: *mut yaml_parser_t, event: *mut yaml_event_t) -> c_int;

#[no_mangle] pub unsafe extern "C" fn yaml_event_delete(
    event: *mut yaml_event_t);

#[no_mangle] pub unsafe extern "C" fn yaml_emitter_initialize(
    emitter: *mut yaml_emitter_t) -> c_int;

#[no_mangle] pub unsafe extern "C" fn yaml_emitter_delete(
    emitter: *mut yaml_emitter_t);

#[no_mangle] pub unsafe extern "C" fn yaml_emitter_set_output_string(
    emitter: *mut yaml_emitter_t,
    output: *mut c_uchar, size: size_t,
    size_written: *mut size_t);

#[no_mangle] pub unsafe extern "C" fn yaml_emitter_set_output_file(
    emitter: *mut yaml_emitter_t, file: *mut FILE);

#[no_mangle] pub unsafe extern "C" fn yaml_emitter_emit(
    emitter: *mut yaml_emitter_t, event: *mut yaml_event_t) -> c_int;

#[no_mangle] pub unsafe extern "C" fn yaml_emitter_flush(
    emitter: *mut yaml_emitter_t) -> c_int;
```

```rust
#[no_mangle] pub extern "C" fn yaml_get_version_string()
    -> *const c_char;

#[no_mangle] pub extern "C" fn yaml_get_version(
    major: *mut c_int, minor: *mut c_int, patch: *mut c_int);
```

A.2 Workspace layout (target)

```
safe-libyaml/ # cargo workspace root
|- Cargo.toml # [workspace] members = [...]
|- safe-libyaml/ # the safe core crate
| |- Cargo.toml
| |- src/
| | |- lib.rs # #![deny(unsafe_code)] etc.
| | |- error.rs # Error, ErrorKind
| | |- event.rs # Event<'a>, OwnedEvent
| | |- token.rs # pub(crate) Token<'a>
| | |- mark.rs # Mark
| | |- encoding.rs # Encoding enum + detection
| | |- parser/{mod,scanner,reader}.rs
| | |- emitter/{mod,writer,style}.rs
| | |- document.rs
| | |- internal.rs
| |- tests/
| | |- api_surface.rs
| | |- yaml_test_suite.rs
| |- benches/parse_benchmark.rs
|- safe-libyaml-sys/ # the C ABI shim
| |- Cargo.toml # crate-type = ["staticlib", "cdylib"]
| |- build.rs
| |- cbindgen.toml
| |- src/lib.rs # extern "C" exports
| |- include/yaml.h # generated (gitignored, regenerated)
|- safe-libyaml-fuzz/ # cargo-fuzz workspace
| |- fuzz_targets/{fuzz_parser,fuzz_emitter,fuzz_roundtrip}.rs
|- safe-libyaml-bench/ # criterion benchmarks
| |- benches/{parse,emit,roundtrip}.rs
```

A.3 Agent task list

The tasks below use the project-mandated checkbox template:

```
[ ] Agent: <name> · Task: <verb-phrase>
Inputs: <files>
```

Output: <crate path / file>
Success: <verifiable criterion>
Est: <hours>

A.3.1 Agent A2 — Scaffold Builder

[ ] Agent: A2 · Task: bootstrap workspace
Inputs: empty repository, project README
Output: Cargo.toml (workspace), rust-toolchain.toml, .gitignore
Success: cargo metadata -no-deps succeeds
Est: 0.5
[ ] Agent: A2 · Task: cargo new -lib safe-libyaml
Inputs: workspace bootstrapped
Output: safe-libyaml/Cargo.toml, src/lib.rs
Success: cargo check -p safe-libyaml succeeds
Est: 0.25
[ ] Agent: A2 · Task: define crate-level lints
Inputs: src/lib.rs
Output: #![deny(unsafe_code, missing_docs, rust_2018_idioms)], lints.toml
Success: cargo check clean; cargo clippy runs
Est: 0.25
[ ] Agent: A2 · Task: define Mark struct
Inputs: knowledge-base mapping table
Output: src/mark.rs with pub struct Mark { index: u64, line: u64, column: u64 }
Success: Mark is Copy + Debug + Clone + Eq + Hash; doctests for ZERO
Est: 0.5
[ ] Agent: A2 · Task: define Encoding enum
Inputs: knowledge-base mapping; libyaml yaml_encoding_t
Output: src/encoding.rs with Any, Utf8, Utf16Le, Utf16Be
Success: repr(u32) matches C order; Default = Any
Est: 0.5
[ ] Agent: A2 · Task: define Error type
Inputs: thiserror crate; libyaml yaml_error_type_e
Output: src/error.rs with Error, ErrorKind, Result
Success: Error: Send + Sync + 'static; variants match knowledge-base spec
Est: 1.0
[ ] Agent: A2 · Task: define ScalarStyle/SequenceStyle/MappingStyle
Inputs: libyaml style enums
Output: src/event.rs (style enums)
Success: repr(u32); serde feature derives
Est: 0.5
[ ] Agent: A2 · Task: define Event<'a> enum
Inputs: this paper's Section 4 spec
Output: src/event.rs with #[non_exhaustive] pub enum Event<'a>
Success: all 11 variants present; Cow<'a, str> for scalar/anchor/tag fields; cargo doc renders cleanly
Est: 1.5

[ ] Agent: A2 · Task: define OwnedEvent and into_owned
Inputs: Event<'a> defined
Output: OwnedEvent, Event::into_owned method
Success: OwnedEvent: 'static + Send + Sync; round-trip test passes
Est: 1.0
[ ] Agent: A2 · Task: define VersionDirective and TagDirective
Inputs: libyaml directive types
Output: src/event.rs
Success: TagDirective<'a> contains handle: Cow<'a, str>, prefix: Cow<'a, str>
Est: 0.5
[ ] Agent: A2 · Task: define internal Token<'a> enum
Inputs: libyaml yaml_token_type_t (22 kinds)
Output: src/token.rs (pub(crate))
Success: 22 variants exhaustively named; not exposed in lib.rs re-exports
Est: 1.0
[ ] Agent: A2 · Task: define Parser<'a> struct stub
Inputs: this paper's Section 3 spec
Output: src/parser/mod.rs with three constructors and parse stub
Success: cargo check passes; bodies are todo!()
Est: 1.0
[ ] Agent: A2 · Task: define Emitter<W: Write> struct stub
Inputs: this paper's API spec
Output: src/emitter/mod.rs with stubs
Success: cargo check passes; Emitter is generic over W
Est: 1.0
[ ] Agent: A2 · Task: define Document and Node types
Inputs: libyaml yaml_document_t, yaml_node_t
Output: src/document.rs
Success: Document owns Vec<Node>; Node is enum {Scalar, Sequence, Mapping}
Est: 1.0
[ ] Agent: A2 · Task: write API surface test
Inputs: scaffold complete
Output: tests/api_surface.rs
Success: tests compile-check that every public type/function exists with the spec'd signature
Est: 1.0
[ ] Agent: A2 · Task: bootstrap safe-libyaml-sys crate
Inputs: workspace
Output: safe-libyaml-sys/Cargo.toml (crate-type = ["staticlib", "cdylib"]), src/lib.rs
Success: cargo build -p safe-libyaml-sys produces libsafe_yaml.{a,so}
Est: 0.5
[ ] Agent: A2 · Task: define #[repr(C)] FFI structs
Inputs: libyaml yaml.h
Output: safe-libyaml-sys/src/lib.rs with yaml_parser_t, yaml_emitter_t, yaml_event_t, yaml_mark_t, yaml_event_type_t, etc.
Success: std::mem::size_of matches the C struct size on the host platform
Est: 2.0

`[ ]` `Agent: A2 · Task: write extern "C" parser exports`
Inputs: safe-libyaml Parser API; libyaml symbol set
Output: `yaml_parser_initialize`, `yaml_parser_delete`, `yaml_parser_set_input_string`, `yaml_parser_parse`, etc.
Success: `nm -D libsafe_yaml.so` contains all 80 expected `yaml_*` symbols
Est: 4.0

`[ ]` `Agent: A2 · Task: write extern "C" emitter exports`
Inputs: safe-libyaml Emitter API; libyaml symbol set
Output: `yaml_emitter_*` extern "C" exports
Success: `nm -D` contains all emitter symbols
Est: 3.0

`[ ]` `Agent: A2 · Task: write cbindgen.toml`
Inputs: this paper's Section 10
Output: `safe-libyaml-sys/cbindgen.toml`
Success: `cbindgen -config cbindgen.toml -o yaml.h` produces a header that diffs cleanly against `libyaml/include/yaml.h` modulo whitespace/comments
Est: 2.0

`[ ]` `Agent: A2 · Task: write build.rs to invoke cbindgen`
Inputs: cbindgen.toml
Output: `safe-libyaml-sys/build.rs`
Success: `cargo build` regenerates `include/yaml.h` on source change
Est: 1.0

`[ ]` `Agent: A2 · Task: write ABI-parity CI check`
Inputs: generated header; reference `libyaml/include/yaml.h`; `libyaml.so`
Output: `.github/workflows/abi-parity.yml`
Success: CI fails if `cbindgen` diff or `nm` symbol diff exceeds the documented allow-list
Est: 1.5

A.3.2 Agent OSG-to-Rust emitter

What is the OSG? The *Ownership Semantic Graph* is a typed intermediate representation produced by the upstream Phase A1 analysis agent. Each node corresponds to a C function or struct field; each node carries (i) an inferred ownership class for every pointer parameter (Owning, BorrowShared, BorrowMut, SharedOwnership, or RawUnsafe), (ii) inferred lifetime constraints between those parameters, and (iii) a confidence score for the inference. The OSG-to-Rust emitter consumes this graph and produces the Rust skeleton signatures into which Phase B agents will fill bodies. The OSG schema and the analysis that produces it are described in the companion paper [12]; for the purpose of this appendix the OSG can be treated as a typed protobuf input whose schema is fixed.

`[ ]` `Agent: OSG-emit · Task: load OSG protobuf`
Inputs: `analysis/osg.pb` produced by Phase A1
Output: in-memory representation with one node per C function
Success: round-trip serialise/deserialise; node count matches function count from `ctags -R`
Est: 2.0

`[ ]` `Agent: OSG-emit · Task: implement signature-emission rule for ownership classes`
Inputs: ownership classification (Owning/BorrowShared/BorrowMut/SharedOwnership)

Output: function that maps OSG node $\mapsto$ Rust signature string

Success: golden tests over ≥ 30 hand-classified libyaml functions agree with the manual port `libyaml-safer`

Est: 4.0

`[ ] Agent: OSG-emit · Task: emit Rust skeleton (one stub per OSG node)`

Inputs: signatures from previous task

Output: `safe-libyaml/src/parser/scanner.rs`, `reader.rs`, `api.rs`, `emitter.rs`, `loader.rs`, `dumper.rs` — all stubs with `todo!()` bodies

Success: `cargo check` passes; one stub per libyaml function ($\sim$175 stubs)

Est: 3.0

`[ ] Agent: OSG-emit · Task: insert lifetime parameters from OSG region inference`

Inputs: OSG region annotations on pointer parameters

Output: lifetime-annotated signatures, e.g. `fn scan(&mut self, buf: &'a [u8]) -> ...`

Success: 90% of inferred lifetimes match the manual port

Est: 5.0

`[ ] Agent: OSG-emit · Task: tag stubs with provenance comments`

Inputs: OSG node IDs

Output: each stub prefixed with `/// libyaml: src/scanner.c:L123 (yaml_parser_scan)`

Success: every stub has a non-empty provenance comment

Est: 1.0

`[ ] Agent: OSG-emit · Task: emit module-level mapping doc`

Inputs: OSG

Output: `docs/translation-map.md` listing C function $\rightarrow$ Rust signature

Success: every libyaml function appears

Est: 1.0

A.3.3 Agent C4 — Idiom Polisher

`[ ] Agent: C4 · Task: configure pedantic clippy lint set`

Inputs: `Cargo.toml`

Output: `lints.toml`, `clippy.toml`

Success: `cargo clippy - -W clippy::pedantic -W clippy::nursery -D warnings` runs and emits the agent's worklist

Est: 1.0

`[ ] Agent: C4 · Task: replace manual index loops with iterators`

Inputs: scanner/parser/emitter source post-Phase B

Output: refactored modules

Success: `clippy::needless_range_loop` count drops to zero; tests still pass

Est: 4.0

`[ ] Agent: C4 · Task: collapse Option::unwrap chains to ? propagation`

Inputs: source

Output: refactored

Success: `grep -c '.unwrap()' src/` below the agreed budget; `cargo test` passes

Est: 3.0

`[ ] Agent: C4 · Task: remove any unsafe blocks remaining in safe crate`

Inputs: source post-Phase B

Output: `#![deny(unsafe_code)]` compiles clean

Success: zero unsafe in `safe-libyaml/src`
Est: 4.0

[ ] `Agent: C4 · Task: add doc comments to every public item`
Inputs: source
Output: doc-commented public API
Success: `#![deny(missing_docs)]` compiles; `cargo doc -no-deps` clean
Est: 4.0

[ ] `Agent: C4 · Task: add #[must_use] to all Result-returning fns`
Inputs: source
Output: annotated
Success: `clippy::must_use_candidate` reports zero remaining
Est: 1.0

[ ] `Agent: C4 · Task: rename single-letter variables`
Inputs: source (post c2rust often uses `i`, `p`, `c`)
Output: renamed
Success: `clippy::min_ident_chars` reports zero (with whitelist for indices in iterator chains)
Est: 2.0

[ ] `Agent: C4 · Task: switch String allocations to Cow at API boundary`
Inputs: `Event<'a>` draft
Output: `Cow<'a, str>` where appropriate
Success: zero-copy benchmark shows $\geq 30\%$ allocation reduction on plain-scalar input
Est: 3.0

[ ] `Agent: C4 · Task: add #[non_exhaustive] to public enums`
Inputs: `Event`, `ErrorKind`, `Encoding`
Output: annotated
Success: cargo-semver-checks reports the next minor as additive-only
Est: 0.5

[ ] `Agent: C4 · Task: review against libyaml-safer for readability parity`
Inputs: both ports
Output: review notes; further refactors
Success: human reviewer judges parity in a sampled ≥ 20 functions
Est: 6.0

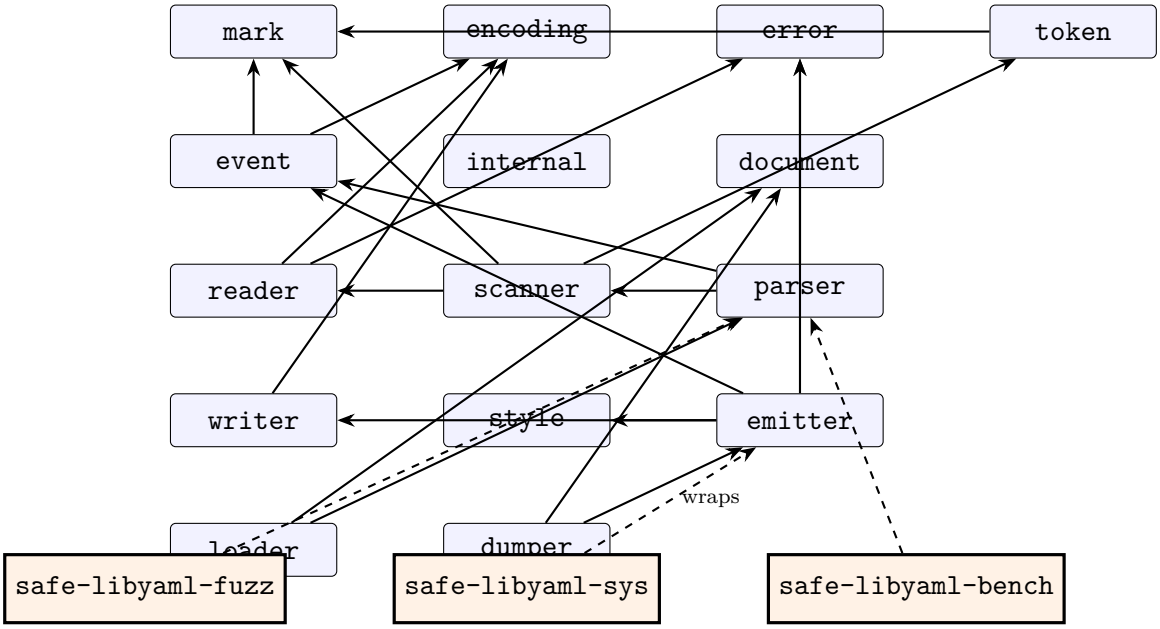
A.3.4 Crate-layout agent (umbrella)

[ ] `Agent: Crate-Layout · Task: bootstrap safe-libyaml-fuzz`
Inputs: workspace
Output: `safe-libyaml-fuzz/` with `cargo-fuzz` configured
Success: `cargo fuzz list` prints `fuzz_parser`, `fuzz_emitter`, `fuzz_roundtrip`
Est: 1.0

[ ] `Agent: Crate-Layout · Task: bootstrap safe-libyaml-bench`
Inputs: workspace
Output: `safe-libyaml-bench/` with criterion benches
Success: `cargo bench` runs three baseline benches
Est: 1.0

A.4 Module dependency diagram

The following diagram shows the workspace's module dependencies. Solid edges denote `use crate::`; dashed edges denote inter-crate `Cargo.toml` dependencies.



Reading. The four "leaf" modules (`mark`, `encoding`, `error`, `token`) have no `safe-libyaml`-internal dependencies and are translated first (Phase B Wave 1). The middle row (`event`, `internal`, `document`) depends only on the leaves. The bottom rows (`reader`/`scanner`/`parser`, `writer`/`style`/`emitter`, `loader`/`dumper`) form the streaming pipeline. The three sister crates (`-sys`, `-fuzz`, `-bench`) depend on the safe core but never on each other.

A.5 Reference `lints.toml`

```
# safe-libyaml/lints.toml -- consumed by C4 (Idiom Polisher)
[lints.rust]
unsafe_code = "deny"
missing_docs = "deny"
rust_2018_idioms = "deny"

[lints.clippy]
pedantic = { level = "warn", priority = -1 }
nursery = { level = "warn", priority = -1 }
# Project-specific allows (justify each in PR description)
module_name_repetitions = "allow"
missing_errors_doc = "allow" # covered by Error type docs
must_use_candidate = "warn"
needless_range_loop = "deny"
```

A.6 Hand-off to Phase B

The artifacts produced by the agents in this appendix — the workspace, the crate skeletons with `todo!()` bodies, the FFI shim with stub bodies delegating to the safe crate, and `cbindgen.toml` producing a parity header — form the input contract for Phase B (the per-function TRANSLATE-TEST-FIX loop) which is described in the companion `c-rust-transpiling` paper. Phase B agents replace each `todo!()` with a real implementation while preserving every type signature emitted by the OSG-to-Rust emitter, ensuring the type-contract discipline mandated by the supervisory `CLAUDE.md` rules.