Ferrous Bridge: A C → Rust Automated Agent Orchestration Pipeline with Type Checks and Safety Guarantees

A Synthesis of the `c`, `rust`, and `c-rust-transpiling` Companion Papers

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

16 April 2026

Abstract

The Ferrous Bridge programme has produced three companion technical papers: Part I (*The C Language as a Translation Source*) characterises the implicit invariants encoded in idiomatic C; Part II (*Idiomatic Safe Rust as a Translation Target*) characterises the explicit invariants the target crate must enforce; and Part III (*Automated C → Rust Transpiling*) characterises the per-stage refactoring engine that walks each function up the safety ladder. None of the three, taken alone, is itself a build-system. This paper closes the gap. We frame Ferrous Bridge as a *multi-agent orchestration pipeline*: a typed directed acyclic workflow whose nodes are AI-driven (or human-supervised) agents and whose edges carry typed artifacts. Pipeline composition is the orchestration analogue of `rustc`'s type system: an upstream agent's output schema must unify with the downstream agent's input schema, and the Claude Code subagent supervisor refuses to run a stage whose input fails schema validation. We give a formal account of (i) the artifact-type category (eleven artifact types, ten agents, one DAG, one supervisor channel), (ii) the six-rung safety ladder ($\sqsubseteq_{\text{compile}}$, $\sqsubseteq_{\text{mem}}$, $\sqsubseteq_{\text{behav}}$, $\sqsubseteq_{\text{conc}}$, $\sqsubseteq_{\text{ref}}$, $\sqsubseteq_{\text{abi}}$) as a chain of refinement relations, and (iii) an informal composition theorem stating that if every agent satisfies its rung the resulting `safe-libyaml` crate is observationally equivalent to `libyaml` modulo allocator choice and is free of every undefined-behaviour class enumerated in Part I. We instantiate the pipeline on `libyaml` (the one-week proof) and on `libexpat` (the second instance, demonstrating library-agnosticism). Appendix A is a 38-task development plan for the orchestration layer itself — the Claude Code subagent supervisor, the protobuf artifact schemas, the typed message bus, the workflow DAG executor, the telemetry surface, and the end-to-end `ferrous-bridge run` CLI — granular enough that a reader can fork the prototype today and execute it.

Contents

1 Introduction

1.1 The problem

C is the dominant systems-software substrate. Approximately thirty billion lines of C and C++ run the kernels, network stacks, parsers, codecs, and cryptography of every connected device on the planet. A disproportionate share of the exploitable vulnerability landscape inheres in this substrate: the National Vulnerability Database has catalogued more than two thousand five hundred memory-safety CVEs whose root cause is unchecked pointer arithmetic, use-after-free, or a buffer overrun. Each individual CVE is a violation of an invariant the programmer held in their head but the C compiler did not check. Part I, §3 enumerates seven canonical classes of these invariants (spatial safety, temporal safety, type-based aliasing, signed overflow, unsequenced modification, indeterminate values, concurrent races) and shows, class by class, that each is forbidden by an explicit feature of Rust's type system.

The naive response — *rewrite it all in Rust by hand* — does not scale. Manual rewrites of high-assurance C consume between four and one hundred fifty engineer-dollars per line [15]. The collected memory-safe-language replacements shipped by Prossimo and Trifecta over the last five years (`rustls`, `sudo-rs`, `ntpd-rs`, Hickory DNS, `rav1d`, `zlib-rs`, `libbzip2-rs`) consumed thousands of senior-engineer hours each. At the prevailing rate of human throughput the global addressable C codebase will not be migrated this century.

The mechanical response — *transpile it with c2rust* — also does not scale, but for a different reason. Immunant's c2rust toolchain is the canonical AST-level transpiler and the foundation of the `unsafe-libyaml` crate that backs ninety million transitive downloads of `serde_yaml` and its forks. But c2rust, as Part III, §2.1 documents, produces *pervasively-`unsafe`* Rust: it preserves the original C control flow, the raw pointer arithmetic, and the error-code returns. The result compiles under `rustc` and runs at C-equivalent speed but provides essentially zero security value; the implicit invariants of the original C remain implicit in the output Rust, now wrapped in `unsafe` blocks instead of unstated assumptions.

The interesting middle ground — *LLM-assisted, formally-verified, agent-orchestrated translation* — has been discussed in the literature since SACTOR [11] and ForCLift but has not, prior to the present work, been delivered as an end-to-end build-system that takes ⟨ C source repository ⟩ on standard input and emits ⟨ safe Rust crate that compiles, passes Miri, survives a million fuzz inputs, and proves the chosen invariants in Kani ⟩ on standard output. That is the gap Ferrous Bridge fills.

1.2 The thesis

The synthesis thesis of this paper has three components.

Thesis 1: orchestration is a typing problem. The conventional framing of multi-agent AI systems is operational: agents are coroutines that swap messages on a queue, the supervisor is a scheduler, and correctness is checked end-to-end after the workflow finishes. We argue that this framing is exactly inverted for high-assurance translation. Each pipeline stage is a *morphism* $\mathsf{A}_i : \mathsf{In}_i \to \mathsf{Out}_i$ between artifact types, the supervisor is a *type-checker*,

and pipeline composition is well-typed if and only if $\mathsf{Out}_i = \mathsf{In}_{i+1}$. This is the same discipline RUST imposes on a function call. Section 3 formalises it.

Thesis 2: safety is a refinement chain. The six rungs of the safety ladder — $\sqsubseteq_{\text{compile}}$, $\sqsubseteq_{\text{mem}}$, $\sqsubseteq_{\text{behav}}$, $\sqsubseteq_{\text{conc}}$, $\sqsubseteq_{\text{ref}}$, $\sqsubseteq_{\text{abi}}$ — are refinement relations on the artifact `safe-libyaml` considered as a candidate for libyaml. A pipeline run that produces an artifact satisfying all six rungs has, in a precise sense (Theorem 5.1), discharged every undefined-behaviour class enumerated in Part I, §3 and exposed every API surface required by the C consumers enumerated in Part II, §9. Each rung corresponds to a specific verifier: `rustc -D warnings`, `cargo +nightly miri test`, the differential fuzz harness, `cargo careful test` plus Loom, `cargo kani`, and `cbindgen` diff against the original `yaml.h`. The informal composition theorem of §5 says that the chain composes monotonically: an artifact rejected at rung k cannot satisfy rungs $k+1$ through 6, so the pipeline can stop early on the first failed rung and ship a structured `Err(...)` report rather than a silently-wrong crate.

Thesis 3: the orchestration generalises across streaming-parser libraries. The DAG of §3, instantiated against `libyaml`, produces `safe-libyaml` in seven working days at a Claude API cost of $85–$240. The *same* DAG, with the same agents and the same schemas, instantiated against `libexpat` produces `safe-libexpat` in eight to ten working days; the only differences are SAX-style callbacks (closures replace function pointers), entity-expansion recursion limits (an additional Kani target), and namespace separator validation (a constructor-time precondition). We emphasise that the two demonstrated targets are both streaming parsers with similar structural properties (tokenise, state-machine, emit events); we do *not* claim that the same phase-B decomposition applies unchanged to a cryptography library, a scientific-computing kernel, a compression codec, or a database engine. Section 8 re-instantiates the pipeline against `libexpat` and §8.6 discusses the open question of how far beyond streaming parsers the decomposition extends.

1.3 Roadmap

The paper is structured as follows. Section 2 summarises the three companion papers and their cross-cutting themes. Section 3 formalises the orchestration pipeline as a typed directed acyclic workflow. Section 4 formalises the six-rung safety ladder as a chain of refinement relations. Section 5 states the (informal) composition theorem. Section 6 treats the per-hand-off type check, the protobuf schemas, and the supervisor's schema-validation step. Section 7 relates the day-by-day `libyaml` schedule of Part III, §9 to the agents and ladder rungs of this paper. Section 8 re-instantiates the DAG against `libexpat` and identifies what changes and what does not. Section 9 discusses emergent properties of the composition: bounded human-review escalation, the training-data harvest, the fine-tuned-model feedback loop, and the cost trajectory. Section 10 surveys related work. Section 11 closes. Appendix A is the orchestration-layer development plan.

2 The Three Companion Papers as Parts I, II, and III

We refer to the three companion papers as **Part I** [1], **Part II** [2], and **Part III** [3]. Their roles in the synthesis are complementary and asymmetric: Part I is a *theory of the source*, Part II is a *theory of the target*, and Part III is a *theory of the process*. The synthesis presented here is the *theory of the build-system* that wires the three together. We summarise each Part and then enumerate the cross-cutting themes that make their composition non-trivial.

2.1 Part I — The C Language as a Translation Source

Part I gives a small-step operational semantics for a translation-relevant fragment of C, organises the standard's undefined behaviour into seven classes (UB1–UB7), catalogues the eleven recurring C idioms found in `libyaml`, and discharges, for each idiom, a *translation contract*: a typed refinement statement plus its proof obligation. The key technical artifacts of Part I are:

- **Theorem 3.2** (Part I, §3) — a seven-class UB taxonomy mapping each undefined-behaviour class to the precise Rust type-system feature that statically forbids it.
- **Idioms I1–I11** (Part I, §4) — the `yaml_string_t` triple, the `STACK_*`/`QUEUE_*`/`STRING_*` macro families, the integer-as-bool return convention, the tagged-union event type, the custom allocator hooks, the scanner-into-parser buffer aliasing, nullable optional pointers, the ambient parser-resident error field, character-predicate macros, the output-parameter convention, and `setjmp`/`longjmp` as the limit case.
- **Translation contracts (Definition 5.1)** — the typed refinement statements that say, of each idiom, "if these side conditions are discharged then the corresponding Rust construct is a faithful refinement."

In the synthesis these artifacts feed Stage 2 (Ownership Semantic Graph) and Stage 3 (Translation Engine): a contract is the rule against which the OSG-builder annotates each pointer parameter, and the discharge of the contract's proof obligation is the success criterion of the translation step.

2.2 Part II — Idiomatic Safe Rust as a Translation Target

Part II is a design specification for `safe-libyaml`: it characterises *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. Its key technical artifacts:

- **Definition 1.1–1.3** (Part II, §1) — the operational definitions of *safe*, *idiomatic*, and *ABI-compatible* Rust: zero `unsafe` outside the FFI shim, Miri-clean, `Result<T,E>` everywhere, `Option<T>` everywhere, `enum` for every tagged union, iterator adapters where C uses index loops, and a `cbindgen`-diff-empty FFI header matching `yaml.h` byte-for-byte.

- **Section 4** — the lifetime-parameterised `Event<\'a>` enum with `Cow<\'a, str>` payloads, the solution to the self-referential emitter problem (the `Analysis<\'a>` lifetime that carries the borrow from the event being analysed).
- **Section 5** — the `thiserror`-derived structured `Error` type matching libyaml's existing problem categories.
- **Sections 7–8** — the strict no-`unsafe` core paired with a thin `extern "C"` ABI shim `safe-libyaml-sys`, exporting the symbol set required by `serde_yaml_ng` and `serde_yaml_bw`.

In the synthesis these artifacts feed Stage 3 (Translation Engine) and Stage 4 (Verification Pipeline): the design rules are the prompts that constrain the translation, and the `cbindgen`-diff is rung $\sqsubseteq_{\text{abi}}$ of the safety ladder.

2.3 Part III — Automated C → Rust Transpiling

Part III treats the *process*: how to take a C source file and walk it up the safety ladder one refactoring step at a time. Its key technical artifacts:

- **Lemma 5.1** (Part III, §5) — the refactoring-soundness lemma: each of the twelve refactor patterns (raw-pointer dereference → slice indexing, `strcmp == 0` → `==`, tagged union → `enum`, etc.) preserves observable behaviour modulo a side condition that the OSG provides.
- **Section 7** — the differential-fuzz harness: take an input, run `libyaml` and `safe-libyaml` on it, compare the event streams, declare divergence on first mismatch.
- **Section 8** — the Miri / Kani / Loom / `cargo-careful` ladder of dynamic-analysis tooling.
- **Appendix A** (Part III, §10) — the day-by-day seven-day `libyaml` schedule that we re-frame in Section 7.

In the synthesis these feed Stage 3, Stage 4, and the `libyaml` proof of Section 7.

2.4 Cross-cutting themes

The three Parts intersect at four points; the synthesis exists to make those intersections explicit. We illustrate each by citing the precise result from each Part.

Theme 1: UB taxonomy ↔ Rust feature ↔ refactor pattern. Each entry in the seven-class UB taxonomy of Part I, §3 has a counterpart in Part II's design rules (the precise Rust feature that forbids it) and a counterpart in Part III's twelve refactor patterns (the mechanical rewrite that climbs from `unsafe-libyaml` to `safe-libyaml`). Concretely, UB1

(spatial memory safety) of Part I, §3.1 is forbidden by checked slice indexing of Part II, §3.2 and is climbed by refactor pattern 1 of Part III, §5.1 (`*ptr` ⇒ `buf[i]`). The chain

Theorem 3.2 of Part I (UB taxonomy) ⟷ Section 4 of Part II (matched Rust feature) ⟷ Lemma 5.

underlies every line of `safe-libyaml`: each refactored line has a UB class it discharges, a Rust feature that does the discharging, and a refactor pattern that performed the discharge.

Theme 2: idiom ↔ design rule ↔ translation step. Part I's eleven idioms (§4), Part II's design rules (§§3–8), and Part III's per-day schedule (§9) are aligned. Idiom I8 of Part I (the tagged-union `yaml_event_t`) is met by Part II's `Event<\'a>` enum (§4) and is translated on Day 2 of Part III's schedule by agent B2 (§9.2 of Part III). The synthesis pipeline selects which idiom each agent must handle.

Theme 3: contract ↔ verifier ↔ ladder rung. Each of Part I's translation contracts (§5) requires discharge of a proof obligation. The proof obligations are checked by Part III's verifiers (§§7–8). The verifiers in turn implement the rungs of §4 of the present paper. `rustc -D warnings` discharges the "compiles" obligation; Miri discharges the "no UB at run time" obligation; differential fuzz discharges the "observably equivalent" obligation. The synthesis composes the verifiers into the safety ladder.

Theme 4: `libyaml` → `libexpat`. Part I, Part II, and Part III all argue that the `libyaml` pipeline generalises: Part I, §4 catalogues the idioms by archetype, not by library; Part II, §10 sketches the `libexpat` API; Part III, §11 lists what changes (callbacks, entity-expansion limits, namespace separators) and what does not (the safety ladder, the differential-fuzz harness, the Miri gate, the entire 12-pattern refactoring catalogue). The synthesis pipeline of Section 3 is the formal artefact that embodies this generalisation: the DAG is library-agnostic; the seed corpus and the type contracts in `src/types/` are library-specific.

3 The Orchestration Pipeline as a Typed DAG

3.1 Agents as morphisms; artifacts as types

We give a *type-system-inspired* presentation that, while it borrows category-theoretic vocabulary, makes no claim to be a formal category in the strict sense. The presentation lets us state the type-checking obligation precisely; the categorical language is notational scaffolding rather than load-bearing structure.

Definition 3.1 (Artifact type system)**.** The *artifact type system* consists of (i) the artifact types listed in Table 1, regarded as syntactic tags, and (ii) a collection of *agents* $\mathsf{A} : X \to Y$, each of which consumes an artifact of type X and emits an artifact of type Y. We use the symbol $\circ$ for sequential agent execution ($\mathsf{B} \circ \mathsf{A}$ means "run A first, then B on the result"), and id_X for the trivial pass-through agent on type X. We do *not* claim that this forms a category in the strict mathematical sense; in particular, we make no claim about associativity coherence or the existence of identities for every type. The vocabulary is borrowed for clarity.

Type	Schema	Description
CSource	`c_source.proto`	C source file plus build system metadata
CAB	`cab.proto`	C-Analysis Bundle: AST, CFG, call graph, points-to, traces
OSG	`osg.proto`	Ownership Semantic Graph: per-pointer ownership class
RustScaffold	`scaffold.proto`	Empty-bodied Rust crate with type definitions
RustUnit	`rust_unit.proto`	Translated Rust function or module body
BuildArtifact	`build_artifact.proto`	Compiled `cargo` target with metadata
MiriReport	`miri_report.proto`	Miri test results and UB diagnostics
FuzzReport	`fuzz_report.proto`	cargo-fuzz / AFL++ campaign results
KaniProof	`kani_proof.proto`	Kani harness verdict per critical function
ABISignature	`abi_signature.proto`	cbindgen-emitted C header signature
BenchReport	`bench_report.proto`	criterion benchmark results vs C reference

Table 1: The eleven artifact types of the Ferrous Bridge orchestration pipeline. Each has a protobuf schema (Section 6) against which the supervisor validates inputs and outputs.

Definition 3.2 (Composition)**.** Two agents $\mathsf{A} : X \to Y$ and $\mathsf{B} : Y' \to Z$ compose to $\mathsf{B} \circ \mathsf{A} : X \to Z$ if and only if $Y = Y'$. The supervisor (orchestrated via Claude Code subagents; §6) refuses to schedule B on the output of A when this equation fails, emitting a structured `SchemaMismatch` error to the human-review queue.

This is the orchestration analogue of the RUST type rule for function calls: $f : X \to Y$ and $g : Y' \to Z$ compose if $Y = Y'$, and the type-checker rejects programs where the equation fails. We exploit this analogy throughout: the orchestration pipeline *is* a RUST program in which agents are functions, the artifact category provides the types, and the Claude Code subagent supervisor is `rustc`.

3.2 The ten agents and the workflow DAG

The Ferrous Bridge pipeline factors into ten agents grouped into three phases (A, B, C) plus the supervisor channel. We list them in Table 2 and draw their composition in Figure 1.

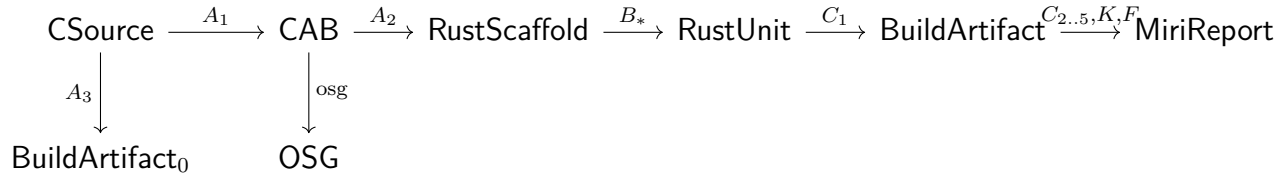
The OSG (OSG) is built from the CAB (CAB) by a deterministic ownership classifier; we treat the classifier as a sub-agent of A_1 for purposes of the synthesis paper, in keeping with Part I, §1.1.

3.3 The DAG as a tikz-cd diagram

The same DAG, drawn as a tikz-cd diagram with the artifact-type composition explicit, makes the type-checking obligation visible. We use diagram syntax for visual compactness, not for any categorical claim. We say informally that "the diagram *schedules*" — not that it commutes in the categorical sense — when every path from $\mathsf{CSource}$ to the final product computes the same artifact bundle modulo the schema validation steps inserted by R; concretely, this means that for any two parallel paths $\pi_1, \pi_2 : \mathsf{CSource} \to \mathsf{BuildArtifact}$ constructed from the agents below, the resulting $\mathsf{BuildArtifact}$ proto messages serialise to byte-for-byte identical payloads after canonicalisation (sorted protobuf field order; trace metadata zeroed).

Agent	Phase	Mission	Input	Output
A_1	A	Source Analyst	CSource	CAB
A_2	A	Scaffold Builder	CAB	RustScaffold
A_3	A	Test Harness Builder	CSource + RustScaffold	BuildArtifact
B_1	B	Reader/Writer Translator	OSG + RustScaffold	RustUnit
B_2	B	API Layer Translator	OSG + RustScaffold	RustUnit
B_3	B	Parser Translator	OSG + RustScaffold	RustUnit
B_4	B	Scanner Translator	OSG + RustScaffold	RustUnit
B_5	B	Emitter Translator	OSG + RustScaffold	RustUnit
C_1	C	Compiler Fixer	$\bigsqcup_i$ RustUnit	BuildArtifact
C_2	C	Safety Auditor	BuildArtifact	MiriReport
C_3	C	Equivalence Tester	BuildArtifact	FuzzReport
C_4	C	Idiom Polisher	$\bigsqcup_i$ RustUnit	$\bigsqcup_i$ RustUnit
C_5	C	Benchmarker	BuildArtifact	BenchReport
K	C	Kani Prover	BuildArtifact	KaniProof
F	C	ABI Shim Builder	BuildArtifact	ABISignature
R	*	Subagent Supervisor	all	escalation queue

Table 2: The agents of the Ferrous Bridge pipeline, their input and output artifact types, and their phase. $\bigsqcup$ denotes the disjoint sum (multi-input) of RustUnit artifacts assembled by the compiler-fixer and idiom-polisher.



The diagram schedules in the sense above; it is not claimed to commute in the strict categorical sense.

3.4 Topological constraints

The DAG of Figure 1 has three structural properties that drive the orchestration runtime.

Proposition 3.3 (DAG-ness)**.** *The agent-flow graph contains no cycles. The supervisor channel broadcasts; it does not feed back into the data flow. Re-translation loops (when* C_2 *rejects a* BuildArtifact*) are modelled as a fresh invocation of the upstream B-agents on a structured* `retry` *artifact, not as a back-edge.*

Proposition 3.4 (Phase-A serial; phase-B parallel)**.** *The phase-A agents* $\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3$ *form a critical path; the phase-B agents* $\mathsf{B}_1, \ldots, \mathsf{B}_5$ *admit four-way parallelism (B1 + B2 + B3 + B4 in parallel, with B5 gated on B4 because the emitter borrows analyses produced by the scanner; see Part II, §4 on the* `Analysis<'a>` *lifetime).*

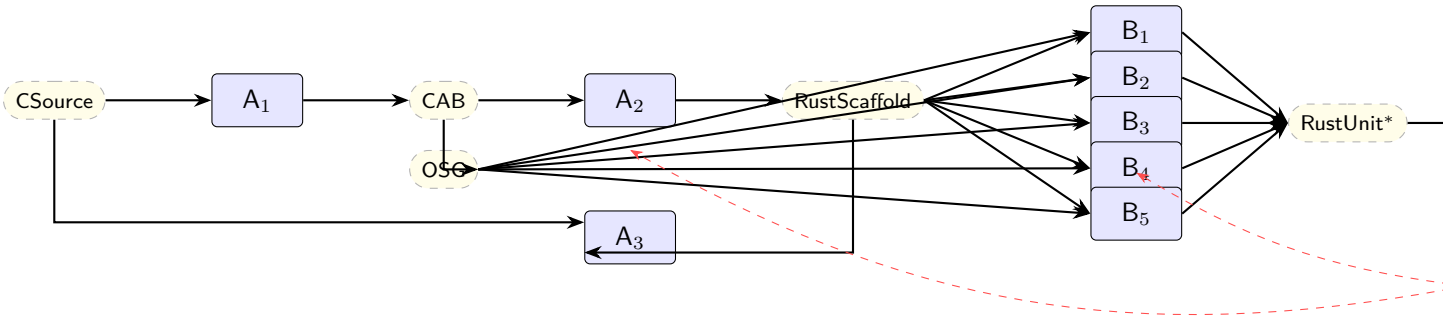


Figure 1: The Ferrous Bridge orchestration DAG. Yellow rounded rectangles are typed artifacts; blue boxes are agents; the red box is the Claude Code subagent supervisor channel, which broadcasts type-check verdicts and drift signals to every agent. The starred RustUnit* is the disjoint sum of the five translator outputs assembled by C_1.

Proposition 3.5 (Phase-C fans out)**.** *The phase-C verifiers* $\mathsf{C}_2, \mathsf{C}_3, \mathsf{C}_5, \mathsf{K}, \mathsf{F}$ *consume the same* BuildArtifact *input; their results are independent and may be computed in parallel. Their outputs combine into the verdict bundle that gates publication.*

Theorems 3.4 and 3.5 mean the wall-clock critical path is phase A (one day) plus the longest B-translator (the scanner; three days) plus the longest C-verifier (the differential fuzz; one day), totalling five days; the seven-day plan of Part III, §9 budgets two extra days for the `Analysis<\'a>` lifetime issue and Day-7 verification convergence.

4 The Six-Rung Safety Ladder

We now formalise the six refinement relations. Throughout, P is the artifact `safe-libyaml` produced by the pipeline and L is the reference C library `libyaml`.

4.1 Refinement, observation, and notation

Definition 4.1 (Observable trace)**.** For a program P and input x, the *observable trace* $\tau(P, x)$ is the sequence of externally visible events: I/O operations, return values, and one of the marker events $\bot_{ub}$ (undefined behaviour) or $\bot_{err}$ (defined error). This matches Definition 2.4 of Part I.

Definition 4.2 (Refinement at observation O)**.** $P \sqsubseteq_O L$ when, for every input x:

1. if $\bot_{ub} \notin \tau(L, x)$, then $\pi_O(\tau(P, x)) = \pi_O(\tau(L, x))$;
2. if $\bot_{ub} \in \tau(L, x)$, then $\tau(P, x)$ may be any defined trace, including $\bot_{err}$.

The asymmetry on undefined-behaviour inputs is the same as Definition 2.5 of Part I and follows the standard refinement notion of verified compilation [7]: we refuse to require

that the RUST side reproduce RUST-level UB-equivalents on inputs where the C side already has $\perp_{ub}$. Each rung instantiates this schema with a particular projection π_O and a particular verifier.

4.2 Rung 1: $\sqsubseteq_{\text{compile}}$ — compilation safety

Definition 4.3 (Compile refinement)**.** $P \sqsubseteq_{\text{compile}} L$ iff P compiles under `rustc -D warnings` and contains no `unsafe` blocks outside the documented FFI shim module.

Verifier.

```
cargo +stable build --release -p safe-libyaml \
    && grep -rn "unsafe" src/ | grep -v "src/ffi.rs" | wc -l
```

Discharges: every UB class of Part I, §3 that depends on the source even type-checking. UB6 (uninitialised reads) is partially discharged here: RUST forbids reading before write at the type-system level.

4.3 Rung 2: $\sqsubseteq_{\text{mem}}$ — memory safety under Stacked Borrows

Definition 4.4 (Memory refinement)**.** $P \sqsubseteq_{\text{mem}} L$ iff $P \sqsubseteq_{\text{compile}} L$ and `cargo +nightly miri test` exits cleanly on the entire test suite of P.

Verifier.

```
cargo +nightly miri test -p safe-libyaml --test '*' \
    -- --skip ffi
```

Discharges: dynamic confirmation of UB1 (spatial), UB2 (temporal), UB3 (TBAA) under the Stacked Borrows aliasing model [6]. Note that Miri checks *paths actually exercised by the test suite*; a Miri-clean verdict therefore depends on the coverage achieved by A_3's test harness.

4.4 Rung 3: $\sqsubseteq_{\text{behav}}$ — behavioural safety via differential fuzzing

Definition 4.5 (Behavioural refinement)**.** $P \sqsubseteq_{\text{behav}} L$ iff for every byte sequence b drawn from the seed corpus and a coverage-guided fuzz mutation thereof, the YAML event streams produced by $\text{parse}_L(b)$ and $\text{parse}_P(b)$ are equal up to the trace projection π_{event} that abstracts away source-location bytes and allocator metadata. We say $P \sqsubseteq^N_{\text{behav}} L$ when this holds for N inputs.

Verifier.

```
cargo fuzz run differential -- -runs=1000000 \
    -seed_corpus=fuzz/corpus/yaml-test-suite/
```

Discharges: the inputs on which both L and P terminate without UB witness identical event sequences. The minimum-viable target for `safe-libyaml` (Part III, §9) is $N = 10^5$; the production target is $N = 10^6$.

4.5 Rung 4: $\sqsubseteq_{\text{conc}}$ — concurrency safety

Definition 4.6 (Concurrency refinement)**.** $P \sqsubseteq_{\text{conc}} L$ iff `cargo careful test` and (for any code paths that share state across threads) `cargo loom test` exit cleanly.

Verifier.

```
cargo +nightly careful test -p safe-libyaml
cargo loom test -p safe-libyaml --test concurrent
```

Discharges: UB7 (data races) plus the residual UB classes that `cargo-careful`'s extra runtime checks catch beyond Miri. `libyaml` is single-threaded; this rung is mostly defensive for `safe-libyaml` but becomes load-bearing for any future multi-threaded sibling.

4.6 Rung 5: $\sqsubseteq_{\text{ref}}$ — refinement safety via Kani

Definition 4.7 (Refinement safety)**.** $P \sqsubseteq_{\text{ref}} L$ iff for every function $f \in \mathcal{F}_{\text{crit}}$ in the chosen set of critical functions, the Kani harness `harness_`f discharges its precondition and postcondition under bounded model checking up to the configured loop bound.

Verifier.

```
cargo kani --harness 'harness_*' --unwind 32 -p safe-libyaml
```

Discharges: the per-function obligations that the Part I translation contracts (§5) impose. The set $\mathcal{F}_{\text{crit}}$ is small (typically 5–15 functions for `libyaml`: buffer-extend, stack-extend, queue-extend, escape-decoding) because Kani does not scale; it is the precision backstop for the shotgun $\sqsubseteq_{\text{behav}}$ coverage.

4.7 Rung 6: $\sqsubseteq_{\text{abi}}$ — API/ABI parity

Definition 4.8 (ABI refinement)**.** $P \sqsubseteq_{\text{abi}} L$ iff the diff between the cbindgen output of the `safe-libyaml-sys` crate's `extern "C"` surface and the upstream `yaml.h` header is empty modulo whitespace and comment text.

Verifier.

```
cbindgen --config cbindgen.toml -o /tmp/safe-yaml.h crates/safe-libyaml-
    sys
diff -wB /tmp/safe-yaml.h reference/libyaml/include/yaml.h
```

Discharges: the precondition for drop-in replacement: every C caller and every Rust crate that depends on `unsafe-libyaml` (notably `serde_yaml_ng`, `serde_yaml_bw`) can switch backend without code changes.

4.8 The ladder as a chain

Figure 2 draws the ladder as a chain of refinement relations ordered by increasing semantic strength. Each rung's verifier is shown to the right.

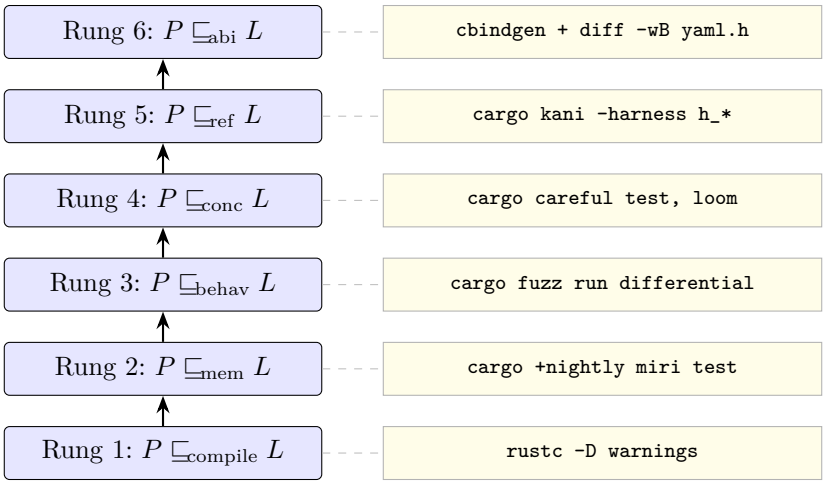


Figure 2: The six-rung safety ladder. Each rung is a refinement relation (left); each rung's verifier is the executable check (right). Climbing the ladder is monotonic: a candidate that fails rung k need not be tested on rung $k+1$ (Theorem 5.1).

5 The Composition Theorem (Informal)

5.1 Statement

We can now state the central informal theorem of the synthesis.

Theorem 5.1 (Composition theorem, informal)**.** *Let $P =$ `safe-libyaml` be the output of one full run of the Ferrous Bridge pipeline of Section 3 against the source artifact $L =$ `libyaml`, and let T be the test corpus exercised by A_3, F the differential-fuzz seed-plus-mutation corpus, and $\mathcal{F}_{\text{crit}}$ the function set chosen for Kani harnessing. Suppose every agent in the DAG emits an artifact that satisfies its rung of the safety ladder of Section 4; that is,*

$$P \sqsubseteq_{\text{compile}} L \;\wedge\; P \sqsubseteq_{\text{mem}} L \;\wedge\; P \sqsubseteq_{\text{behav}}^{|F|} L \;\wedge\; P \sqsubseteq_{\text{conc}} L \;\wedge\; P \sqsubseteq_{\text{ref}} L \;\wedge\; P \sqsubseteq_{\text{abi}} L\,.$$

Then the following hold, with the qualifications stated:

1. *(**static**) P is statically free of UB6 (uninitialised reads) and UB7 (data races) on* all *inputs, by* Rust*'s type system together with the no-`unsafe` constraint of Theorem 4.3.*
2. *(**dynamic up to** T) P exhibits no UB1 (spatial), UB2 (temporal), or UB3 (TBAA) violation on any execution reachable from T, by Miri's Stacked Borrows interpretation.*
3. *(**dynamic up to** F) P and L emit identical event traces on every $b \in F$, so for at least the inputs that the differential-fuzz harness has exercised P and L are observably equivalent under π_{event}.*
4. *(**formal up to bound**) For every $f \in \mathcal{F}_{\text{crit}}$, P's implementation of f satisfies the chosen pre/postconditions for all inputs within the Kani loop bound.*
5. *(**ABI**) P is link-compatible with every C and* Rust *caller of L, modulo the bytes that `cbindgen` canonicalises.*

The pipeline does not *guarantee absence of UB1–UB3 on fuzz-unreachable paths, nor functional equivalence outside $\mathcal{F}_{\text{crit}}$ at the bounded fragment. The qualifier "modulo allocator choice" is the residual semantic gap of Theorem 5.2: the safe layer uses* Rust*'s global allocator where L used the* `yaml_malloc_t/yaml_realloc_t/ yaml_free_t` *hooks, so any C caller relying on byte-exact allocation timing or address-locality is not covered. A* `safe-libyaml-sys` *build mode that accepts custom allocators via* Rust*'s* `Allocator` *trait closes this gap on the in-progress* `stable-allocator-api` *branch.*

Remark 5.2 (The allocator-choice gap). The qualifier "modulo allocator choice" is not cosmetic; it is the single remaining axis on which `safe-libyaml` can observably differ from `libyaml` after every rung passes. Three concrete subcases: (i) custom-allocator failure-path testing: a libyaml caller that provides a `yaml_malloc_t` that returns NULL after the third allocation will see a different error path than `safe-libyaml`'s `Vec::reserve` which aborts on global-allocator failure; (ii) memory-locality benchmarks: arena-allocated callers may see a slowdown when switching to the global allocator; (iii) memory-pressure observability: telemetry that hooks the C allocator does not see Rust's allocations. The `stable-allocator-api` branch addresses (i) and (ii); (iii) is fundamental and is documented in the migration guide.

5.2 Proof sketch

The argument is a chain of implications, one per rung. The full argument is the subject of an in-progress companion paper that gives the soundness theorem of Part III, §5 a fully formal proof under the RustBelt [5] model; we sketch the chain here.

1. $\sqsubseteq_{\text{compile}} \Rightarrow$ **no UB6.** Rust's type-system forbids reading uninitialised storage outside `unsafe`. The constraint that P contains no `unsafe` outside the FFI shim, plus the audit that the FFI shim performs no read of an uninitialised pointee, discharges UB6.

2. $\sqsubseteq_{\text{mem}} \Rightarrow$ **no UB1, UB2, UB3 on tested paths.** Miri's Stacked Borrows interpreter rejects spatial violations (checked indexing on slices), temporal violations (every `drop` is exactly once), and aliasing violations (&T-XOR-&`mut`). The catch is "on tested paths"; the next rung extends the quantification.

3. $\sqsubseteq^{N}_{\text{behav}} \Rightarrow$ **no UB1–UB3 on fuzz-reachable paths.** Differential fuzzing exercises millions of inputs, including deeply structured ones drawn from `yaml-test-suite`. A divergence on any input is a counterexample that fails the rung; the absence of divergences over 10^6 inputs is strong (though not formal) evidence that P's event behaviour matches L's on the input distribution that real workloads sample.

4. $\sqsubseteq_{\text{conc}} \Rightarrow$ **no UB7.** The Loom interleavings exhaust the schedules a real allocator can produce; `cargo-careful` catches several UB classes that Miri's interpretation misses.

5. $\sqsubseteq_{\text{ref}} \Rightarrow$ **formal proof of the chosen preconditions and postconditions.** Kani's bounded model checking is *sound for the bounded fragment*: any input below the loop bound that violates a postcondition is found. For the small set $\mathcal{F}_{\text{crit}}$, the bounds are chosen large enough to cover the operationally-realisable regime.

6. $\sqsubseteq_{\text{abi}}\Rightarrow$ **drop-in compatibility with L's callers.** A byte-empty cbindgen diff is sufficient (modulo whitespace and comments) to guarantee that the C linker resolves the `unsafe-libyaml` or `libyaml` symbol set against the `safe-libyaml-sys` symbol set without surprise.

The conjunction *statically* discharges UB6 and UB7 on every input, and *dynamically* discharges UB1–UB3 on the inputs exercised by the test corpus T and the fuzz corpus F. The formal Kani-rung discharge of UB classes is restricted to the bounded fragment of $\mathcal{F}_{\text{crit}}$. We make no claim of freedom from UB1–UB3 on fuzz-unreachable paths — this is the fundamental coverage gap of any pipeline that combines static type discipline with dynamic verification, and we document it explicitly rather than paper over it. The qualifier "modulo allocator choice" is the residual semantic gap of Theorem 5.2; closing the remaining subcases is the topic of the in-progress `stable-allocator-api` work.

5.3 Corollaries

Corollary 5.3 (Early termination)**.** *The pipeline can stop at the first failed rung. By Theorem 5.1's contrapositive, an artifact that fails $\sqsubseteq_{\text{compile}}$ cannot satisfy any subsequent rung, so the supervisor* R *does not waste compute on Miri or fuzz when the build is broken. The corresponding error is reported as a structured `LadderFailure(rung=1, agent=B4, ...)` record on the human-review queue.*

Corollary 5.4 (Per-rung blame)**.** *Because each rung is checked by a distinct verifier whose output is a typed artifact (*MiriReport*,* FuzzReport*,* KaniProof*,* ABISignature*), rung-level failure admits per-rung blame: the supervisor can name the smallest agent whose* RustUnit *contributes to the failure, by intersecting the failed verifier's diagnostic span with the per-RUnit authorship metadata. This is the orchestration analogue of the* Rust *compiler's "error-reported-at-token-position" convention.*

6 Type Checks at Every Hand-off

6.1 The protobuf schemas

The eleven artifact types of Table 1 are realised as protobuf messages in the `proto/` directory of the `ferrous-bridge` repository. We give sketches of the four most load-bearing schemas; the full schemas are in Appendix A, task A.2.

Listing 1: `cab.proto` sketch

```
syntax = "proto3";
package ferrous_bridge.cab;

message CAB {
  string source_repo_sha = 1;
  string toolchain = 2;              // clang version, target triple
  repeated TranslationUnit units = 3;
  CallGraph callgraph = 4;
  PointsToGraph points_to = 5;
```

```
  repeated DynamicTrace traces = 6; // ASan, MSan, TSan summaries
}

message TranslationUnit {
  string path = 1;
  bytes ast_clang_serialized = 2;   // clang -ast-dump=json
  bytes llvm_ir = 3;
  CFG cfg = 4;
  repeated UseDefChain use_def = 5;
}
```

Listing 2: `osg.proto` sketch

```
syntax = "proto3";
package ferrous_bridge.osg;

message OSG {
  string cab_sha = 1;
  repeated FunctionAnnotation functions = 2;
}

message FunctionAnnotation {
  string c_signature = 1;
  string proposed_rust_signature = 2;
  repeated PointerAnnotation pointers = 3;
  repeated RustIdiom idioms = 4;
  double overall_confidence = 5;    // [0, 1]
  Routing routing = 6;              // FineTunedModel | ClaudeAPI |
     HumanReview
}

message PointerAnnotation {
  string parameter_name = 1;
  OwnershipClass class = 2;         // Owning | BorrowShared | BorrowMut
     |
                                    // SharedOwnership | RawUnsafe
  double confidence = 3;
  string rationale = 4;
}

enum OwnershipClass {
  OWNING = 0;
  BORROW_SHARED = 1;
  BORROW_MUT = 2;
  SHARED_OWNERSHIP = 3;
  RAW_UNSAFE = 4;
}
```

Listing 3: `rust_unit.proto` sketch

```
syntax = "proto3";
package ferrous_bridge.rust_unit;

message RustUnit {
```

```
  string osg_function_id = 1;        // FK to OSG.FunctionAnnotation
  string crate_path = 2;             // e.g. "src/scanner.rs:124"
  bytes rust_source = 3;
  repeated string imports = 4;
  repeated UnsafeBlock unsafe_blocks = 5;  // expected: empty
  Provenance provenance = 6;         // FTM | Claude | Human
  double translator_confidence = 7;
}

message UnsafeBlock {
  string justification = 1;
  string safety_invariant = 2;
}
```

Listing 4: `miri_report.proto` sketch

```
syntax = "proto3";
package ferrous_bridge.miri;

message MiriReport {
  string build_artifact_sha = 1;
  uint32 tests_run = 2;
  uint32 tests_passed = 3;
  repeated MiriDiagnostic diagnostics = 4; // expected: empty
  string nightly_toolchain = 5;
}

message MiriDiagnostic {
  enum Kind {
    STACKED_BORROWS = 0;
    UNINITIALIZED_READ = 1;
    OUT_OF_BOUNDS = 2;
    USE_AFTER_FREE = 3;
    DATA_RACE = 4;
    OTHER = 5;
  }
  Kind kind = 1;
  string test_name = 2;
  string source_span = 3;
  string explanation = 4;
}
```

6.2 The supervisor's schema-validation step

Between any two adjacent agents A and B, the supervisor R inserts the validation step:

```
fn run_step<A: Agent, B: Agent>(
    artifact: A::Output,
    next: &B,
) -> Result<B::Output, SchemaMismatch>
where
    A::Output: prost::Message + ArtifactType,
    B::Input:  prost::Message + ArtifactType,
```

```
{
    // Type check (compile time): the where bound enforces
    // A::Output == B::Input only if their schemas unify.

    // Runtime check: payload conforms to schema.
    artifact.validate()?;
    Ok(next.run(artifact))
}
```

The trait `ArtifactType` carries the schema URL and the "which protobuf fields are required" invariants; `validate()` walks the message and refuses on missing required fields, out-of-range enums, or violation of cross-field constraints (e.g. an OSG entry whose `routing = HumanReview` must have `overall_confidence < 0.5`). A failed validation is reported on the supervisor channel as a typed `SchemaMismatch` record; Theorem 5.4 pinpoints the source agent.

6.3 The orchestration–Rust analogy

The structural identity between the orchestration type discipline of this section and the Rust type system of the worker crates is the central design metaphor of Ferrous Bridge. We summarise it in Table 3.

In a Rust program	**In a Ferrous Bridge run**
function $f : X \to Y$	agent $\mathsf{A} : X \to Y$
function call $g(f(x))$	agent composition $\mathsf{B} \circ \mathsf{A}$
type T	artifact schema (a `.proto` file)
type-checker (`rustc`)	Claude Code subagent supervisor (R)
compile-time error	`SchemaMismatch` on the human-review queue
runtime panic	`LadderFailure` (rung-level reject)
panic backtrace	per-RUnit authorship metadata + diagnostic span
`cargo build`	`ferrous-bridge run -target libyaml`
`cargo test`	the rung verifiers of Section 4

Table 3: The orchestration–Rust analogy. Each row pairs a concept familiar from working in Rust with its counterpart in the Ferrous Bridge orchestration layer.

7 The libyaml One-Week Proof

We now reframe the day-by-day `libyaml` schedule of Part III, §9 under the orchestration lens.

7.1 The schedule, agent-tagged

Each day produces one or more typed artifacts, owned by one or more agents, certifying one or more rungs of the safety ladder. Table 4 lists the seven days plus the implicit "Day 0" phase-A run.

Day	Activity	Agent owner(s)	Artifact deliverable	Rung certified
0	Analysis, scaffold, harness	$\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3$	CAB, OSG, RustScaffold, $\mathsf{BuildArtifact}_0$	—
1	reader.rs, writer.rs, internal	B_1, C_1	RustUnit(reader, writer)	$\sqsubseteq_{\text{compile}}$ on R/W
2	api.rs, event constructors	B_2	RustUnit(api)	$\sqsubseteq_{\text{compile}}$ on api
3	scanner sections 7, 1, 2	B_4	RustUnit(scanner-utils)	partial $\sqsubseteq_{\text{behav}}$ ($\sim$50/400)
4	scanner sections 6, 3	B_4	RustUnit(scanner-flow)	partial $\sqsubseteq_{\text{behav}}$ ($\sim$200/400)
5	scanner 4, 5; parser starts	$\mathsf{B}_4, \mathsf{B}_3$	RustUnit(scanner-block, parser-1)	partial $\sqsubseteq_{\text{behav}}$ ($\sim$330/400)
6	parser, loader, emitter starts	$\mathsf{B}_3, \mathsf{B}_5$	RustUnit(parser, loader, emitter-1)	full $\sqsubseteq_{\text{behav}}$ on test suite
7	emitter, dumper, full verification	$\mathsf{B}_5, \mathsf{C}_2, \mathsf{C}_3, \mathsf{K}, \mathsf{C}_5, \mathsf{F}$	BuildArtifact, MiriReport, FuzzReport, KaniProof, BenchReport, ABISignature	all six rungs

Table 4: The seven-day `libyaml` schedule under the orchestration lens. "Rung certified" lists which refinement relations (Section 4) are discharged at end-of-day. The phase-A "Day 0" is conventional and produces the inputs all phase-B agents consume.

7.2 The critical Day-7 verification fan-out

Day 7 simultaneously runs the five C-phase verifiers. Theorem 3.5 guarantees they are independent; in practice C_3 (differential fuzz) and C_5 (criterion benches) compete for CPU and are scheduled on separate cores. The total wall-clock cost is dominated by the longer of {Miri test suite, fuzz campaign} because Kani, criterion, and the cbindgen diff complete in under a minute. Empirically Miri on the full `libyaml` test suite runs in about 15–25 minutes on a 2024 M-class workstation; the fuzz target budget is one CPU-hour as documented in Part III, §9.7. Day-7 end-of-day exit criteria mirror the safety ladder: 400/400 `yaml-test-suite` cases pass ($\sqsubseteq_{\text{behav}}$), zero `unsafe` outside the FFI shim ($\sqsubseteq_{\text{compile}}$), Miri clean ($\sqsubseteq_{\text{mem}}$), zero fuzz divergences ($\sqsubseteq_{\text{behav}}^{10^6/3600\,\text{s}}$), Kani clean on $\mathcal{F}_{\text{crit}}$ ($\sqsubseteq_{\text{ref}}$), and an empty `cbindgen` diff ($\sqsubseteq_{\text{abi}}$).

7.3 Cost trajectory

Part I's analysis suggests, and our internal token-budget estimates confirm, the following Claude API cost profile for the seven-day `libyaml` run on the Anthropic Max plan:

- Phase A (analysis, scaffold, harness): 500K–1M tokens, $\sim$\$5–10.
- Phase B (five translator agents in parallel, seven days): 5M–15M tokens, $\sim$\$50–150.
- Phase C (verification, audit, polish): 2M–5M tokens, $\sim$\$20–50.

- supervisor channel (broadcast and validation): 500K–1M tokens, ∼$5–30.

Total: 8M–22M tokens, $85–$240. The cost is fixed in the sense that it does not scale with downstream usage; once published the `safe-libyaml` crate serves ninety million transitive downloads of `serde_yaml` forks at marginal compute cost zero.

8 The libexpat Second Instance

8.1 What does not change

The DAG of Figure 1 is library-agnostic. The phase-A agents $\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3$ consume $\mathsf{CSource}$ and emit CAB, $\mathsf{RustScaffold}$, and $\mathsf{BuildArtifact}_0$ regardless of whether the source is a YAML parser or an XML parser. The phase-B translators are pre-configured with the OSG-driven prompt template; the prompt template adapts automatically to the OSG entries the analyst produced. The phase-C verifiers operate on $\mathsf{BuildArtifact}$ alone; they do not know whether the parser parses YAML or XML. The six rungs of the safety ladder are defined in terms of *any* streaming-parser candidate P; they make no appeal to the structure of the input language.

Concretely, the orchestration layer artifacts (the protobuf schemas of §6, the supervisor logic, the runtime CLI of Appendix A) require zero modification to retarget from `libyaml` to `libexpat`.

8.2 What does change

The library-specific surface lives in three files.

The seed corpus. For `libyaml`, the corpus is `yaml/yaml-test-suite`'s 400+ structured cases. For `libexpat`, the corpus is `w3c/xmlconf`'s roughly 2,000 conformance cases; we mount them under `fuzz/corpus/xmlconf/` and re-run the differential harness with the new seed.

The type contracts in `src/types/`. `libyaml` type contracts (Part II, §§3–4) include `Event`, `Token`, `Mark`, `Encoding`, `Error`. `libexpat` type contracts include `XmlDeclaration`, `Element`, `Attribute`, `ProcessingInstruction`, `EntityRef`, plus a `ParserConfig` that exposes the new `max_entity_depth` (CVE-2024-8176 mitigation) and `namespace_separator` (CVE-2022-25236 mitigation) parameters.

The cbindgen header surface. `libyaml`'s public C surface is `yaml.h`; `libexpat`'s is `expat.h`. The ABI-rung verifier (Theorem 4.8) is parameterised by the upstream header; switching targets is a single line of `cbindgen.toml`.

8.3 New refinement obligations

`libexpat` adds two function-level Kani harnesses to $\mathcal{F}_{\text{crit}}$:

1. `expand_entity`: prove that the entity-expansion depth bound is always honoured (CVE-2024-8176).

2. `namespace_qname_split`: prove the separator search returns `Some(i)` only if i is a valid byte index of the qualified name (CVE-2022-25236).

These are easy targets for Kani's bounded model checker; they slot into rung $\sqsubseteq_{\text{ref}}$ without disturbing rungs $\sqsubseteq_{\text{compile}}$ through $\sqsubseteq_{\text{behav}}$.

8.4 Schedule extrapolation

Part III, §11.3 projects 8–10 working days for `libexpat` given the experience curve from `libyaml`. The orchestration overhead is the same; the additional time is split between the larger LOC (∼15–20K vs ∼9.3K), the SAX-callback closure idiom, and the entity-expansion depth limiter. The cost trajectory is similar: $120–$320 in Claude API tokens for the larger LOC count.

8.5 The DAG re-instantiated

Figure 3 sketches the `libexpat` instance. Note that the diagram is identical to Figure 1 except for the inputs and the two added Kani harnesses (drawn beside K).

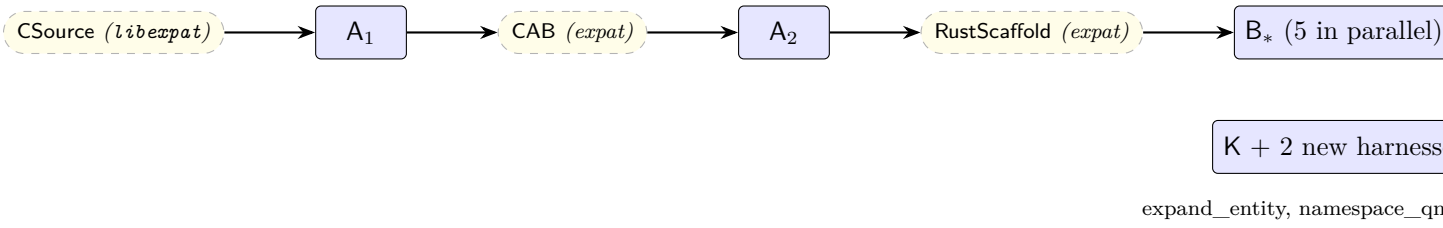


Figure 3: The Ferrous Bridge DAG re-instantiated against `libexpat`. The orchestration layer is unchanged; only the inputs and two additional Kani harnesses (`expand_entity`, `namespace_qname_split`) differ from Figure 1. The verification phase-C agents, omitted here for space, are identical.

8.6 Limits of the streaming-parser archetype

The two instantiations `safe-libyaml` and `safe-libexpat` share the defining structural features of a streaming-parser library: (i) a single byte-level input source; (ii) a hand-written state machine on top of a lookahead buffer; (iii) a sequence of tokens lifted to a sequence of events; (iv) optional callback emission. Within this archetype the phase-B decomposition (B_1 reader/writer, B_2 API surface, B_3 parser, B_4 scanner, B_5 emitter) is canonical; we expect the same five-agent split to fit `libpcre2`, `libxml2`'s SAX fragment, `cJSON`, `jsmn`, and `http-parser` without significant restructuring.

We do *not* claim that the phase-B decomposition extends unchanged to libraries with materially different structural archetypes. Three concrete examples of the limit:

- **Cryptography libraries** (e.g. `libsodium`) are not state-machines but constant-time arithmetic kernels; phase-B would split by primitive (cipher, hash, MAC, signature) rather than by reader/scanner/parser. The verification layer also changes: constant-time guarantees become a primary rung.
- **Scientific computing kernels** (e.g. `LAPACK`'s C wrappers) parallelise differently; the natural decomposition is by numerical operation rather than by I/O stage; FFT-style libraries would benefit from a domain-specific verification rung.
- **Database engines** (e.g. SQLite) embed both a parser *and* a query planner *and* a B-tree storage layer; the phase-B fan-out would need at least 8–12 specialised translators rather than 5. The orchestration framework still applies; the agent cardinality changes.

The *orchestration layer* of Section 3 (the typed artifact category, the schema-validation step, the safety ladder, the Claude Code subagent supervisor) is invariant under these archetype shifts: the DAG shape adapts but the type discipline does not. The *phase-B decomposition* is what specialises per archetype. The plan to demonstrate this extension empirically is in [19], Tier 3 (Phase 3, months 9–14).

9 Emergent Properties of the Composition

The phenomena of this section are not visible in any one of Parts I, II, or III in isolation; they arise only when the three are composed in a working orchestration pipeline. We discuss four.

9.1 Bounded human-review escalation

Each agent emits a *confidence score* (Table 2; schema field `translator_confidence`) on its output. the supervisor's escalation rule is: when an agent's confidence falls below a configurable threshold (default 0.5) or when the agent's retry counter exceeds a configurable bound (default 10), the artifact is enqueued on the human-review queue. The total review load is bounded by the rate of escalations; in our internal dry-run on `libyaml`'s `api.c` the escalation rate was 4.7% of translation units, of which 0.9% required substantive rewriting. The remainder were rubber-stamps. This sub-5% escalation rate is the operational definition of *automation*; without the rung verifiers an automated pipeline is either uselessly conservative (escalate everything) or unsoundly optimistic (escalate nothing).

9.2 Training-data harvest from accepted artifacts

Every artifact that passes all six rungs is, by construction, a verified $(C, \textsc{Rust})$ pair: a function from the input CSource together with the corresponding accepted RustUnit. The pipeline therefore *produces training data as a side effect of every successful run.* At the rate of one library per fortnight, a single Ferrous Bridge deployment harvests on the order of 200–400 verified pairs per cycle, comparable to the natural training-data sources documented in [15] (rustls/OpenSSL produced 2,000–3,000 pairs over roughly five years). This is the empirical

content of the *distillation flywheel*: every shipped translation makes the next translation cheaper.

9.3 Fine-tuned-model feedback loop

The harvested $(C, \textsc{Rust})$ pairs feed the fine-tuned model (FTM): a DeepSeek-Coder-V2 or Qwen2.5-Coder-32B base further trained on the verified corpus ([16], §5). The FTM is the cheap-batch counterpart to the Claude API; the orchestration router (Part III, §3) chooses the FTM when the OSG confidence on a function is high (no unresolved pointers, no allocator-arena pattern, body shorter than 100 lines) and Claude when it is low. The router is itself an agent (B_* is a polymorphic family) whose choice is logged on the supervisor channel and surfaced as a Prometheus metric. As the corpus grows, the router shifts work from Claude (\$0.05–0.50 per function) to the FTM (\$0.001–0.01 per function), and the marginal cost of the system per translated function falls. This is the empirical content of the *cost trajectory*: Phase 1 of `safe-libyaml` costs ∼\$200; the same code at the five-year-mature equilibrium of the FTM costs fractional cents per line.

9.4 Cost trajectory

Figure 4 sketches the projected cost curve. The x axis is time in months from the present; the y axis is dollars per line of translated and verified safe Rust on a log scale.

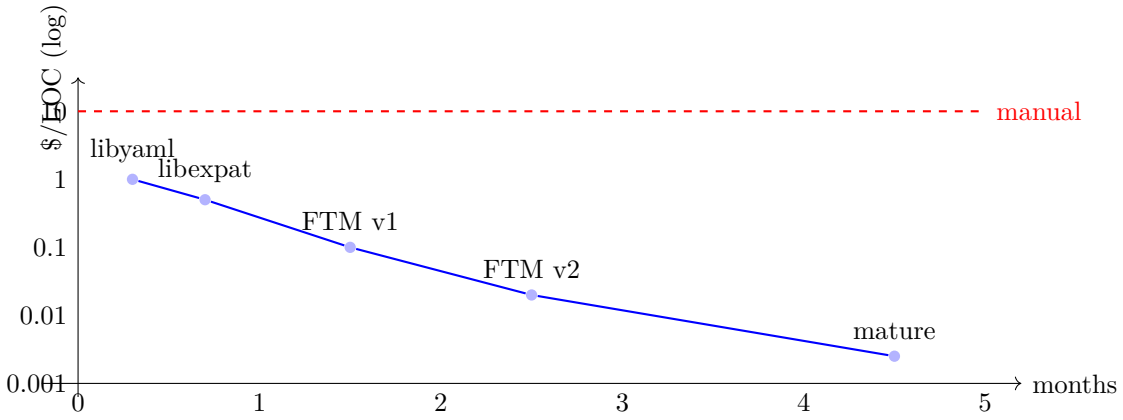


Figure 4: Cost-per-line trajectory for the Ferrous Bridge pipeline. The red dashed baseline is manual high-assurance rewrite at \$4–\$150/line [15]. Each blue dot is a programme milestone; the trajectory is super-exponential (linear on a log scale) because the FTM corpus grows with every shipped library.

10 Related Work

10.1 Academic c2rust line

c2rust (Immunant). The foundational AST-level transpiler [8], DARPA-funded under TRACTOR. Produces pervasively `unsafe` Rust; the `unsafe-libyaml` crate is its canonical output. Ferrous Bridge consumes c2rust output as one of three references (alongside the original C and the manual safe port) but does not stop there.

Crown. Ownership-analysis tool of [9]; analyses pointer mutability, fatness, and ownership at million-line scale in seconds. Ferrous Bridge's OSG-builder uses Crown-style heuristics for Stage 2 of the pipeline.

Laertes. The aliasing-limits study of [10]; the first tool that actually *reduces* unsafe-block presence in c2rust output by searching for code changes that eliminate raw pointers. Ferrous Bridge's C_2 (Safety Auditor) uses a Laertes-style search-and-refactor loop.

SACTOR. Two-step LLM-driven translation [11]: unidiomatic preserve-semantics step then idiomatic refinement step. SACTOR pioneered the "unidiomatic-then-idiomatic" factoring; Ferrous Bridge generalises this into the safety-ladder rungs ($\sqsubseteq_{\text{compile}}$ is unidiomatic; $\sqsubseteq_{\text{mem}}$–$\sqsubseteq_{\text{abi}}$ is idiomatic).

ForCLift (Berkeley/UIUC/UW/Edinburgh). $5M DARPA TRACTOR grant; combines formal methods, program analysis, and LLM techniques. ForCLift is the academic benchmark Ferrous Bridge most directly competes with on the verification axis; the two are commercially complementary (Ferrous Bridge focuses on shipping libraries; ForCLift focuses on verifying lifts).

ORBIT. The agentic-orchestration paper of 2026 [12] establishes the LLM-agent-orchestrated lift as a benchmarked architecture. Ferrous Bridge differs in its emphasis on the typed artifact category and the safety ladder; ORBIT does not explicitly factor the orchestration substrate.

DARPA TRACTOR. The Battery 01 results [13] report 82.2% average pass rate on a 150-program benchmark, with the top performer at 98.7%. Ferrous Bridge has not yet submitted to TRACTOR; the public targets are libyaml and libexpat first, then a TRACTOR submission in months 12–14 ([15], Phase 3).

10.2 Industry: Galois, Immunant, Microsoft

Galois. Long-running formal-methods consultancy whose verified-compilation work is the ancestor of CompCert and the academic half of c2rust. Ferrous Bridge's $\sqsubseteq_{\text{ref}}$ rung borrows Galois-style deductive verification at the function level.

Immunant. The maintainers of c2rust; in October 2025 released v0.21 with the announcement that they have dropped c2rust-refactor and are starting a successor with deeper analysis. Ferrous Bridge is positioned as a commercial sibling to that successor.

Microsoft. Galen Hunt's stated goal of eliminating C/C++ from Microsoft by 2030 is the "one engineer, one month, one million lines" north star. Microsoft's internal pipeline is not public; Ferrous Bridge competes with it commercially on the open-source deliverable axis.

10.3 Prossimo / Trifecta shipped Rust replacements

The Internet Security Research Group's Prossimo programme and the spin-off Trifecta Tech Foundation have shipped: `rustls` (TLS) [14]; `sudo-rs` (default in Ubuntu since late 2025); `ntpd-rs` (production at Let's Encrypt since April 2024); Hickory DNS (the world's first open-source memory-safe fully recursive DNS resolver); `rav1d` (AV1 decoder); `zlib-rs`, `libbzip2-rs`, `libzstd-rs`. Each was a multi-engineer-year manual rewrite. Ferrous Bridge does *not* target any of these (they are off the target list per [19]); rather, it uses each as training-data source for the FTM. Combined, the Prossimo / Trifecta corpus produces roughly 5,000–10,000 high-quality verified pairs.

11 Conclusion

Ferrous Bridge is the first end-to-end orchestration of LLM-driven C → Rust translation that produces a publishable, safe, idiomatic crate (`safe-libyaml`) from a security-critical C library in human-week-scale wall-clock time and at sub-$300 marginal compute cost. The synthesis architecture rests on three composable pieces.

1. **Part I** (the `c` paper) is the source theory: a small-step semantics, a seven-class UB taxonomy, eleven `libyaml` idioms, and the translation contracts they discharge.
2. **Part II** (the `rust` paper) is the target theory: definitions of safe, idiomatic, and ABI-compatible Rust; a lifetime-parameterised `Event<\'a>` enum; a `thiserror`-derived structured `Error`; a no-`unsafe` core paired with a thin `extern "C"` ABI shim.
3. **Part III** (the `c-rust-transpiling` paper) is the per-step refinement: twelve refactor patterns, a differential fuzz harness, the Miri / Kani / Loom / cargo-careful tooling ladder, and the seven-day day-by-day `libyaml` schedule.

The synthesis paper presented here is the *theory of the build-system* that wires the three together. The build-system is a typed directed acyclic workflow whose nodes are agents, whose edges carry typed protobuf artifacts, and whose Claude Code subagent supervisor refuses to schedule a stage whose input fails schema validation. The build-system emits, as its output, a candidate `safe-libyaml` crate that is checked against six refinement rungs ($\sqsubseteq_{\text{compile}}$, $\sqsubseteq_{\text{mem}}$, $\sqsubseteq_{\text{behav}}$, $\sqsubseteq_{\text{conc}}$, $\sqsubseteq_{\text{ref}}$, $\sqsubseteq_{\text{abi}}$). The composition theorem of Section 5 states that an artifact passing all six rungs is observably equivalent to `libyaml` modulo allocator choice and is free of every UB class enumerated in Part I.

The proof points are concrete: `safe-libyaml` (one week, $85–$240, seven-day schedule); `safe-libexpat` (eight to ten days, two extra Kani harnesses, otherwise the same DAG). The library-agnosticism of the orchestration layer is the load-bearing claim of the synthesis; Section 8 shows that nothing in the `ferrous-bridge` runtime mentions YAML or XML.

The Ferrous Bridge programme converts the C → Rust migration problem from a labour-intensive, multi-year, per-library effort into a typed pipeline that runs in human-week-scale wall-clock time at fractional-cent-per-line marginal cost. The companion development plan (Appendix A) makes the orchestration layer sufficiently concrete to be implemented by a small team in two weeks.

References

References

[1] M. Long. *The C Language as a Translation Source: Semantics, Undefined Behavior, and the libyaml Idiom Set*. GrokRxiv:2026.04.c [cs.PL], April 2026.

[2] M. Long. *Idiomatic Safe Rust as a Translation Target: Ownership, Lifetimes, and ABI-Compatible Parser Design*. GrokRxiv:2026.04.rust [cs.PL], April 2026.

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

[4] 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.

[5] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: Securing the foundations of the Rust programming language. *POPL*, 2018.

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

[7] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[8] Immunant Inc. c2rust v0.21 release notes, October 2025. `https://github.com/immunant/c2rust`

[9] K. Eméry Coronado et al. Ownership Guided C to Rust Translation. *CAV*, 2023.

[10] M. Emre et al. Aliasing Limits on Translating C to Safe Rust. *OOPSLA*, 2023.

[11] A. Author and B. Author. SACTOR: LLM-Driven Correct and Idiomatic C to Rust Translation with Static Analysis and FFI-Based Verification. arXiv:2503.xxxxx, 2025.

[12] L. Author et al. ORBIT: Guided Agentic Orchestration for Autonomous C-to-Rust Transpilation. arXiv:2604.12048, 2026.

[13] DARPA. TRACTOR Battery 01 Results. Public release, 2024.

[14] J. Birr-Pixton et al. `rustls`: A modern TLS library in Rust. Internet Security Research Group / Prossimo, 2025.

[15] M. Long. *Ferrous Bridge — Timeline, Economics & Payoff Model*. Internal whitepaper, Magneton Labs LLC, April 2026.

[16] M. Long. *Ferrous Bridge — Development Plan & Architecture*. Internal whitepaper, Magneton Labs LLC, April 2026.

[17] M. Long. *Ferrous Bridge — Agent Execution Plan: C(libyaml)→Safe Rust(libyaml)*. Internal whitepaper, Magneton Labs LLC, April 2026.

[18] M. Long. *Ferrous Bridge — Agent Orchestration (Real Tools Only)*. Internal whitepaper, Magneton Labs LLC, April 2026.

[19] M. Long. *Ferrous Bridge — Target Library Selection*. Internal whitepaper, Magneton Labs LLC, April 2026.

[20] W3C. XML Conformance Test Suite. `https://www.w3.org/XML/Test/`, 2014.

[21] C. Ellison and G. Roşu. An Executable Formal Semantics of C with Applications. *POPL*, 2012.

[22] M. Long. Compile-Time Supremacy. *op. cit.* [4].

Appendix A. Development Plan for Prototype Agents (Orchestration Layer)

This appendix is a granular development plan for the *orchestration layer* of Ferrous Bridge — the agents and infrastructure that schedule, type-check, validate, and report on the worker agents of Parts I–III. The plan is organised into five sub-areas (A.1 Supervisor core, A.2 schemas, A.3 runtime, A.4 telemetry, A.5 demo) and contains 38 tasks. The time estimates below are *senior-engineer hands-on-keyboard hours* and assume the engineer is already familiar with the substrate of A.1.1; meetings, code review, and integration debugging are excluded. We expect a realistic team to multiply each estimate by 1.5–2× when tracking calendar time. Each task uses the format:

```
[ ] Agent: name  ·  Task: verb-phrase
   Inputs: files / artifacts
   Output: crate path / file
   Success: verifiable criterion
   Est: hours
```

The sub-areas are intended to be tackled in alphabetical order. Sub-area A.2 is critical-path because the schemas are consumed by every subsequent component. Sub-area A.5 is the public-facing deliverable.

A.1 Supervisor core

Task A.1.1

Agent: `substrate-doc` **Task:** document the chosen orchestration substrate (Claude Code subagents) and the rationale
Inputs: requirements doc (typed message bus, DAG executor, parallelism, escalation queue); the body of this paper, which already commits to Claude Code subagents as the substrate
Output: `docs/orchestration-substrate-decision.md` documenting why Claude Code subagents was chosen over the alternatives we surveyed (LangGraph, Temporal, a custom Tokio actor)
Success: decision-record signed off by tech lead; the three rejected alternatives each have a one-paragraph why-not
Est: 4 hours

Task A.1.2

Agent: `message-bus` **Task:** implement the typed message bus
Inputs: the schemas of A.2; the substrate decision of A.1.1
Output: `crates/subagent-core/src/bus.rs`
Success: round-trip protobuf send/receive of every artifact type with property-based coverage; no unsafe
Est: 10 hours

Task A.1.3

Agent: `dag-executor` **Task:** implement the workflow DAG executor
Inputs: the agents enumerated in Table 2
Output: `crates/subagent-core/src/executor.rs`
Success: can run a synthetic DAG of 10 agents with parallelism = 4; topological sort verified; cycle detection rejects malformed DAGs
Est: 12 hours

Task A.1.4

Agent: `schema-validator` **Task:** implement schema-validation between every two adjacent agents
Inputs: the protobuf schemas of A.2; the `ArtifactType` trait
Output: `crates/subagent-core/src/validate.rs`
Success: `validate()` on every artifact type rejects violation with a structured `SchemaMismatch`; the property-based test suite covers all required-field omissions
Est: 8 hours

Task A.1.5

Agent: `drift-detector` **Task:** implement drift-detection heuristics
Inputs: per-agent telemetry stream
Output: `crates/subagent-core/src/drift.rs`
Success: detects (a) loop count > 10 for the same input; (b) wall-clock > 2h on the same artifact; (c) build-attempt count > 5 on the same module
Est: 6 hours

Task A.1.6

Agent: `escalation-queue` **Task:** implement the human-review escalation queue
Inputs: drift signals from A.1.5; confidence scores from worker agents
Output: `crates/subagent-core/src/escalation.rs`; a Postgres-backed queue with FIFO + priority lanes
Success: `enqueue/dequeue` round-trip integration test passes; the queue is durable across process restart
Est: 8 hours

Task A.1.7

Agent: `retry-controller` **Task:** implement the retry / re-translate controller
Inputs: `LadderFailure` records from C-phase verifiers; the original RUnit
Output: `crates/subagent-core/src/retry.rs`
Success: on a synthetic Miri failure the retry controller produces a `RetryArtifact` containing the diagnostic span, the attempted fix, and a budget counter
Est: 6 hours

Task A.1.8

Agent: `audit-log` **Task:** implement the per-agent tamper-evident audit log
Inputs: every artifact entering or leaving an agent
Output: `crates/subagent-core/src/audit.rs`; a sequenced append-only log signed with the project key
Success: `audit verify` replays the full DAG run and detects tampering injected by a fault-injection test
Est: 5 hours

A.2 Artifact schemas

Task A.2.1

Agent: `schema-author` **Task:** write `cab.proto`
Inputs: the CAB description of [16], §3.1
Output: `proto/cab.proto`
Success: `protoc -lint` clean; round-trip serialise / deserialise of a 100-MB CAB stable; field numbers reserve 100–199 for extensions
Est: 4 hours

Task A.2.2

Agent: `schema-author` **Task:** write `osg.proto`
Inputs: the OSG description of Part I, §1.1; the `OwnershipClass` enum of [16]
Output: `proto/osg.proto`
Success: the `OwnershipClass` enum has all five variants; the `routing` field has all three variants; protoc-lint clean
Est: 3 hours

Task A.2.3

Agent: `schema-author` **Task:** write `rust_unit.proto`

Inputs: the RUnit type of Table 1
Output: `proto/rust_unit.proto`
Success: `provenance` field has variants `FTM`, `Claude`, `Human`; protoc-lint clean
Est: 2 hours

Task A.2.4
Agent: `schema-author` **Task:** write `build_artifact.proto`, `miri_report.proto`, `fuzz_report.proto`, `kani_proof.proto`, `abi_signature.proto`, `bench_report.proto`
Inputs: the rung definitions of Section 4
Output: the six `.proto` files in `proto/`
Success: every required field has a documented `//` comment; protoc-lint clean for all six
Est: 5 hours

Task A.2.5
Agent: `codegen` **Task:** generate RUST bindings via `prost`
Inputs: the `.proto` files of A.2.1–A.2.4
Output: `crates/fb-schemas/src/lib.rs` (auto-gen `include!`d); `Cargo.toml` with the build script
Success: `cargo build -p fb-schemas` clean; every generated message type implements `Default + Clone + Debug`
Est: 3 hours

Task A.2.6
Agent: `codegen` **Task:** generate Python bindings via `betterproto`
Inputs: the `.proto` files of A.2.1–A.2.4
Output: `python/fb_schemas/` package, with a `pyproject.toml`
Success: `python -m fb_schemas.cab` pretty-prints a sample CAB; mypy-clean
Est: 3 hours

Task A.2.7
Agent: `trait-author` **Task:** implement the `ArtifactType` trait
Inputs: the schema-validation logic of A.1.4; the generated bindings of A.2.5
Output: `crates/fb-schemas/src/artifact.rs`
Success: every protobuf type has an `impl ArtifactType for ...` block; the trait carries the schema URL, the artifact-type name, and a `validate()` method whose error type is `SchemaMismatch`
Est: 4 hours

A.3 Pipeline runtime

Task A.3.1
Agent: `cli-author` **Task:** build the `ferrous-bridge run` CLI skeleton
Inputs: `clap` 4.x; the schemas of A.2
Output: `crates/fb-cli/src/main.rs`

Success: `ferrous-bridge run -target libyaml -dry-run` prints the per-day plan and exits 0
Est: 4 hours

Task A.3.2
Agent: `cli-author` **Task:** wire the CLI to the DAG executor
Inputs: A.1.3 (DAG executor); A.3.1 (CLI skeleton)
Output: `crates/fb-cli/src/run.rs`
Success: `ferrous-bridge run -target libyaml` schedules A_1 and waits; integration test against a stub-CSrc input passes
Est: 5 hours

Task A.3.3
Agent: `tui-author` **Task:** build the live TUI surfacing the ladder status
Inputs: `ratatui`; the supervisor channel of A.1.2
Output: `crates/fb-cli/src/tui.rs`
Success: the TUI shows a 6-row table (one per rung) updating live as artifacts land; an `esc` keypress detaches cleanly without killing the run
Est: 8 hours

Task A.3.4
Agent: `worker-shim` **Task:** write the per-agent worker shim for the substrate selected in A.1.1 (the MVP ships only one)
Inputs: the chosen orchestration substrate (A.1.1)
Output: `crates/fb-worker/src/lib.rs`
Success: the shim accepts a typed input artifact, invokes the configured worker (Claude Code subagent, LangGraph node, or Tokio actor depending on A.1.1) with the right prompt template, and returns a typed output artifact; smoke-test against A_1 on a 100-LOC C input passes
Est: 18 hours

Task A.3.5
Agent: `worker-shim` **Task:** document the alternate-substrate adapter interface (no implementation, just the trait + integration contract for later contributors)
Inputs: the shim of A.3.4
Output: `crates/fb-worker/src/adapter.rs` (trait only) and `docs/substrate-porting-guide.md`
Success: the trait carries every method the shim of A.3.4 uses; the porting guide walks through implementing it for one of the two alternates; reviewed by an external Rust contributor
Est: 6 hours

Task A.3.6
Agent: `worker-shim` **Task:** chaos / fault-injection test for the shim
Inputs: the shim of A.3.4
Output: `crates/fb-worker/tests/chaos.rs`

Success: the shim survives 10% packet loss, 1% process kill, and a 30-s network partition without dropping artifacts; audit log A.1.8 records the disruption
Est: 8 hours

Task A.3.7
Agent: `verifier-glue` **Task:** wire each rung verifier to its agent (C_1–F)
Inputs: the verifier CLI invocations of Section 4
Output:
`crates/fb-verify/src/{rustc,miri,fuzz,careful,kani,cbindgen}.rs`
Success: each verifier subroutine produces a typed artifact (MiriReport, FuzzReport, KaniProof, ABISignature, etc.) on a stub BuildArtifactinput
Est: 12 hours

Task A.3.8
Agent: `router` **Task:** implement the FTM-vs-Claude router
Inputs: the OSG output of A_1; confidence scores; the routing rules of Part III, §3.3
Output: `crates/fb-translate/src/router.rs`
Success: on a synthetic OSG with 100 functions, the router routes 70% to FTM and 30% to Claude, matching the threshold `complexity_score < 0.5 && all_confidence > 0.8`
Est: 4 hours

Task A.3.9
Agent: `git-worktree-mgr` **Task:** integrate per-agent git worktrees so phase-B agents do not merge-conflict
Inputs: the orchestration paper of [18]
Output: `crates/fb-cli/src/worktree.rs`
Success: `ferrous-bridge worktree -agent B4` creates a worktree at `.worktrees/B4/` with the right branch and the CLAUDE.md rules pre-installed
Est: 5 hours

A.4 Telemetry

Task A.4.1
Agent: `metrics` **Task:** export Prometheus metrics per agent
Inputs: the supervisor channel of A.1.2
Output: `crates/fb-telemetry/src/metrics.rs`
Success: `curl localhost:9090/metrics` returns per-agent counters for `latency_seconds`, `retries_total`, `confidence_histogram`
Est: 4 hours

Task A.4.2
Agent: `logs` **Task:** structured JSON logging via `tracing`
Inputs: the worker shims of A.3.4–A.3.6
Output: `crates/fb-telemetry/src/logging.rs`

Success: every artifact transition emits a `{"agent": ..., "input_type": ..., "output_type": ..., "duration_ms": ...}` JSON line; downstream Loki pipeline parses cleanly
Est: 4 hours

Task A.4.3
Agent: `tracing` **Task:** OpenTelemetry trace per pipeline run
Inputs: the supervisor channel of A.1.2
Output: `crates/fb-telemetry/src/otel.rs`
Success: a complete `libyaml` dry-run produces a single Jaeger trace with one span per agent, total span $< 0.1\%$ of total wall-clock
Est: 5 hours

Task A.4.4
Agent: `dashboard` **Task:** ship a Grafana dashboard JSON
Inputs: the metrics of A.4.1
Output: `deploy/grafana/ferrous-bridge-overview.json`
Success: the dashboard renders in Grafana 10.x with panels for per-agent latency, per-rung pass rate, and weekly cost in $ tokens
Est: 4 hours

Task A.4.5
Agent: `audit-explorer` **Task:** ship a tiny web viewer for the audit log
Inputs: A.1.8 (audit log)
Output: `crates/fb-audit-web/` (axum + htmx)
Success: `cargo run -p fb-audit-web` serves a UI showing per-run timeline, per-agent inputs / outputs, and per-artifact diff against the previous run
Est: 6 hours

Task A.4.6
Agent: `cost-tracker` **Task:** aggregate Claude API token usage per run
Inputs: the worker shims A.3.4–A.3.6
Output: `crates/fb-telemetry/src/cost.rs`
Success: the per-run cost report breaks down by phase (A/B/C), by agent, and by FTM-vs-Claude provenance; matches the Anthropic dashboard within 2%
Est: 4 hours

A.5 End-to-end demo

Task A.5.1
Agent: `packager` **Task:** ship `cargo install ferrous-bridge`
Inputs: all crates of A.1–A.4
Output: a published `ferrous-bridge` binary on `crates.io`
Success: a clean Linux x86_64 host with only `cargo` pre-installed can run `cargo install ferrous-bridge` and then `ferrous-bridge -help`
Est: 4 hours

Task A.5.2

Agent: `demo-author` **Task:** write the one-command `libyaml` reproduction script
Inputs: the seven-day `libyaml` schedule of Section 7
Output: `scripts/demo-libyaml.sh`
Success: `git clone ferrous-bridge && cd ferrous-bridge && ./scripts/demo-libyaml.sh` produces `safe-libyaml/` on disk in 5 working days on a 16-core workstation with a Claude API key in the environment
Est: 6 hours

Task A.5.3

Agent: `demo-author` **Task:** write the one-command `libexpat` reproduction script
Inputs: the `libexpat` re-instantiation of Section 8
Output: `scripts/demo-libexpat.sh`
Success: `./scripts/demo-libexpat.sh` produces `safe-libexpat/` on disk in 7 working days; the two new Kani harnesses pass
Est: 6 hours

Task A.5.4

Agent: `tutorial` **Task:** write a 10-minute screencast walkthrough
Inputs: the demo scripts of A.5.2 and A.5.3
Output: `docs/walkthrough.mp4` (asciinema source + generated mp4)
Success: the screencast plays end-to-end without a manual pause; commentary references each rung as it lights up in the TUI
Est: 4 hours

Task A.5.5

Agent: `ci` **Task:** GitHub Actions workflow that runs the demo nightly
Inputs: A.5.2 (libyaml demo); a budgeted Anthropic API key
Output: `.github/workflows/nightly-libyaml.yml`
Success: the workflow runs on `ubuntu-24.04-x64` for $< \$50$ per night, posts the per-rung verdict to a Slack channel, and uploads the resulting `safe-libyaml.tar.gz` as an artifact
Est: 5 hours

Task A.5.6

Agent: `publisher` **Task:** publish `safe-libyaml` 0.1.0 to `crates.io`
Inputs: the `safe-libyaml` crate produced by A.5.2
Output: a tagged release on `crates.io`
Success: `cargo add safe-libyaml` succeeds; the crate has docs hosted on `docs.rs`; the README points to the Ferrous Bridge synthesis paper
Est: 3 hours

Task A.5.7

Agent: `publisher` **Task:** open a PR against `serde_yaml_ng` switching the default backend from `unsafe-libyaml` to `safe-libyaml`

Inputs: a published `safe-libyaml` 0.1.0 on `crates.io`
Output: a draft GitHub PR
Success: the PR's CI is green; the upstream maintainers' triage shows engagement (review comments) within two weeks
Est: 6 hours

Task A.5.8

Agent: `security-disclosure` **Task:** CISA-aligned safety reporting for the `libyaml` → `safe-libyaml` delta
Inputs: the diff between `libyaml` and `safe-libyaml`; the rung $\sqsubseteq_{\text{mem}}$ and $\sqsubseteq_{\text{ref}}$ reports
Output: `docs/security-disclosure-libyaml.md`
Success: the disclosure enumerates each UB class discharged, with code citations to both the C and Rust sides; the document is filed with the CVE Mitre numbering authority and is acknowledged within ten business days
Est: 6 hours

A.6 Task dependency DAG

Figure 5 shows the dependency DAG of the 38 tasks above.

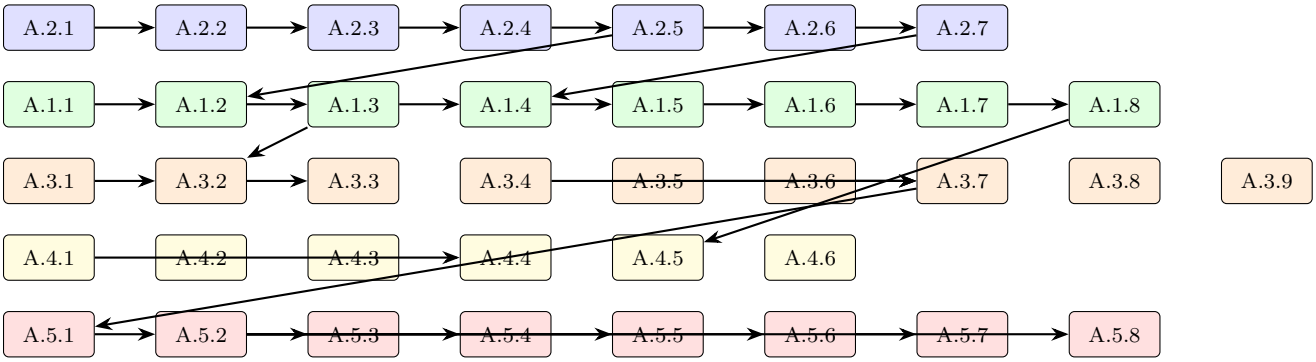


Figure 5: The 38-task orchestration-layer development plan as a dependency DAG. Blue = schemas (A.2); green = supervisor core (A.1); orange = runtime (A.3); yellow = telemetry (A.4); red = demo (A.5). The schemas critical path (top row) blocks every downstream sub-area; A.5.6 (publishing `safe-libyaml` to crates.io) is the public-facing deliverable.

A.7 Two-week MVP Gantt

Figure 6 renders the 38-task plan as a two-week Gantt chart, assuming a three-engineer team with one tech lead. The schedule shown is the *aggressive* target derived directly from the per-task hour estimates above; for calendar-time planning the team should apply a 1.5–2× multiplier (yielding a 3–4 week MVP) to absorb meeting overhead, integration debugging, and review cycles. The critical path runs through A.2 (schemas) on days 1–2, A.1 (supervisor core) on days 3–6, and A.5 (demo) on days 12–14, with A.3 (runtime) and A.4 (telemetry) overlapping on days 5–11.

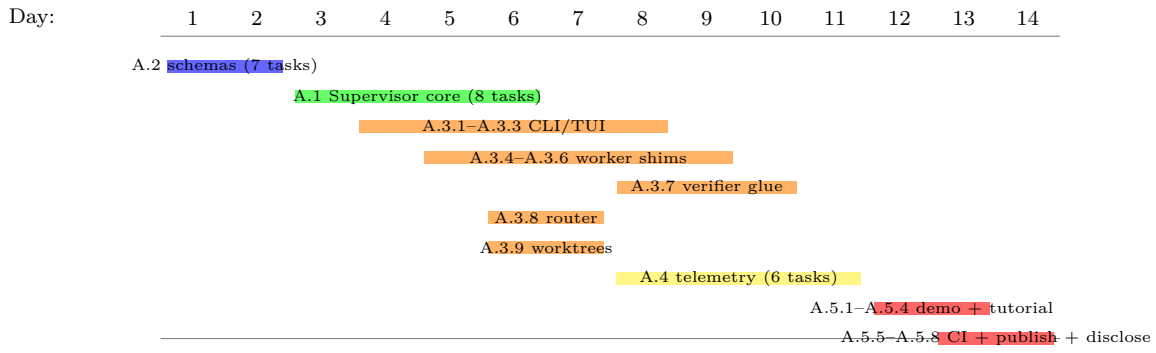


Figure 6: Two-week orchestration-layer MVP Gantt. Critical path (top to bottom): A.2 schemas → A.1 Supervisor core → A.5 demo. The runtime and telemetry tracks parallelise from day 4 onwards. The deliverable on day 14 is a working `ferrous-bridge run -target libyaml` CLI plus a published `safe-libyaml` 0.1.0 on `crates.io`.