

CORAL: Closed-loop Oracle Alignment for Agentic Tax Engine Modernization

Maarten De Raedt Ahmed Abdullah Abhinav Agrawal

Thomson Reuters Labs

{maarten.deraedt, ahmed.abdullah, abhinav.agrawal}@thomsonreuters.com

Abstract

Modernizing legacy software is often framed as code migration, an approach that assumes access to source code and an understanding of system internals. In practice these assumptions fail: source may be inaccessible, data in proprietary formats, and documentation dispersed. We study an underexplored regime in which a replacement is aligned to a legacy system treated as a black-box oracle, using only its input–output behavior.

We present CORAL (Closed-loop Oracle Alignment), an agentic framework that aligns a modern U.S. individual income tax engine (Form 1040) to a production oracle (a system built over decades) by iteratively refining LLM-bootstrapped form schemas. Each round runs a scenario on both systems, compares e-file output field-by-field, and diagnoses discrepancies against an authoritative IRS knowledge base (KB), grounding every fix in published regulation rather than relying on model knowledge. An edit is kept only if re-execution shows no meaningful field-level F_1 regression and all tests pass, a deterministic check rather than model self-assessment. Round-robin sequencing over under-covered forms then reaches the long tail of forms, not just the high-frequency ones a taxpayer shares with everyone.

Across 119 tax-expert-authored scenarios, each a complete taxpayer return, CORAL repairs defects from single-field corrections to reviving entire non-functional forms (53 fixed; 13 escalated to human review, ensuring ambiguous rules and input-data gaps are audited manually), lifting macro- F_1 on a 30-scenario held-out test split, averaged over returns, from an LLM-bootstrapped start (0.17) to 0.90.

1 Introduction

Modernizing legacy software is a common organizational undertaking: to retire years of techni-

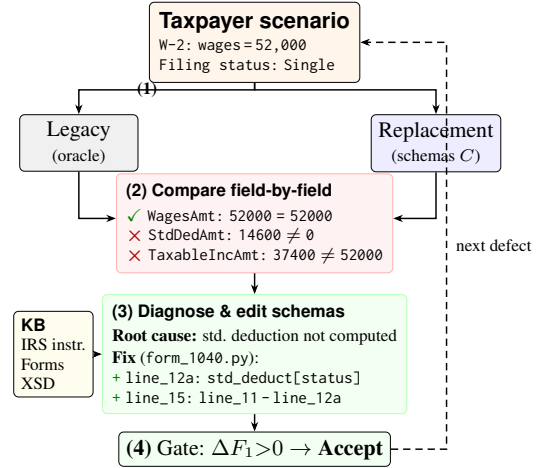


Figure 1: The CORAL loop on a concrete defect: the legacy system applies the standard deduction, the replacement does not. Comparison (2) surfaces the mismatch; the agent edits the schema (3), accepting only if ΔF_1 improves (4).

cal debt, adopt current technology, improve transparency, and, increasingly, to make core systems AI-friendly, such that agents and copilots can sit alongside deterministic business logic instead of bypassing it (Seacord et al., 2003; Khadka et al., 2014). This need is acute in regulated financial domains such as tax, where rules change on a fixed schedule and errors carry compliance risk. Figure 1 previews what a modernization step looks like in our setting: a taxpayer scenario is fed to both the legacy engine and its replacement, the outputs disagree field-by-field, and an agent diagnoses and fixes the mismatch under a deterministic acceptance gate.

The obstacle is understanding the internals of the software itself. Source may be proprietary or no longer available, data sits in dated proprietary formats, authoritative documentation is dispersed rather than centralized, and the original authors

have often moved on. The work also demands sustained specialist developer time, which is in short supply in most organizations (Ziftci et al., 2025; Khadka et al., 2014).

Most current approaches assume this barrier can be crossed. LLM-assisted migration at scale performs source-to-source transformation between readable codebases (Bhatia et al., 2024; Ziftci et al., 2025), and human-in-the-loop agent frameworks coordinate developers with LLMs on enterprise software tasks (Takerngsaksiri et al., 2025). Automated program repair (Weimer et al., 2009; Nguyen et al., 2013) and recent agentic software engineering systems (Jimenez et al., 2024; Yang et al., 2024; Bouzenia et al., 2025; Wang et al., 2025) similarly assume access to a working codebase paired with pre-authored defect reports or test suites. None of these addresses the regime in which the legacy system is effectively a black box yet must be replaced.

A different line of work uses LLMs to recover behavior directly from black-box artifacts: decompiling binaries and synthesizing function-level equivalents (Tan et al., 2024). These techniques are promising but remain at the level of small, isolated units. They have not been stress-tested as the backbone of a real, regulated replacement system built from behavior alone.

We use U.S. individual income tax preparation (Form 1040) as a real-world stress test for this regime, where correctness is a matter of regulatory compliance. A production tax engine receives taxpayer documents and computes a return spanning dozens of interdependent forms whose e-file output must satisfy IRS Modernized e-File (MeF) XML schemas (Internal Revenue Service, 2024b) exactly, under published, deterministic rules defined by authoritative IRS documentation rather than the black-box engine. Since these rules change every filing season, a replacement grounded in IRS documentation is what makes AI-driven year-over-year updates tractable. The same configuration, a black-box legacy engine that is nonetheless queryable and surrounded by authoritative external documentation, appears beyond tax: mainframe-era payroll and regulated financial reporting systems where institutional knowledge has eroded, and healthcare claims engines whose rules are spread across payer contracts and compliance manuals.

We present CORAL, an agentic framework for *oracle-only modernization*: aligning a modern replacement to a legacy system treated strictly as a black-box oracle. In each round, CORAL ex-

ecutes the same scenario on both systems, compares outputs field-by-field, diagnoses each discrepancy against an authoritative IRS knowledge base (form instructions, forms, MeF XSD schemas) such that every repair traces to published regulation, edits declarative form schemas to fix the root cause, and accepts the edit only if a deterministic F_1 gate confirms no meaningful regression under re-execution. A persistent defect store with deduplication, cooldowns, and retry caps prevents oscillation and circular repairs across long runs.

Contributions.

- (i) **Oracle-only modernization of a regulated financial system.** A framework that aligns a modern replacement to a black-box legacy system using only its input–output behavior and *authoritative regulatory documentation* (IRS instructions, forms, MeF schemas), without legacy source or internals, so every repair is grounded in published regulation rather than relying on model knowledge.
- (ii) **Gated agentic loop with persistent defect memory.** A deterministic F_1 gate verifies each edit by re-execution rather than model self-assessment, while a defect store with deduplication, cooldowns, and retry caps prevents oscillation and circular repairs.
- (iii) **Round-robin sequencing over under-covered forms.** Choosing the next form to repair by round-robin over forms below a coverage floor, rather than by whichever edit most raises the per-scenario score, drives repair into the long tail of forms a per-scenario objective would otherwise abandon.
- (iv) **Evaluation on U.S. individual income tax (Form 1040).** From LLM-bootstrapped schemas emitting almost no correct output to 0.90 per-scenario and 0.70 per-form macro- F_1 on a 30-scenario held-out test split (of a 119-scenario, tax-expert-authored corpus), with a taxonomy of the 53 repaired defects and the 13 escalated for human review, so the gap is auditable rather than a silent error.

2 Preliminaries

Tax engine. A tax engine turns taxpayer source documents (W-2s, 1099s, Schedule C data) into a return spanning the interdependent Form 1040

family (Internal Revenue Service, 2024a). Filing requires triggering the right forms (e.g., dividends trigger Schedule B), populating fields from source documents, evaluating status- and threshold-dependent calculations, propagating values across forms, and serializing to MeF XML. A defect at any stage cascades downstream, making manual alignment infeasible at scale.

Notation. Let O be a black-box oracle and S_C a target system parameterized by mutable codebase C . Both map an input $x \in \mathcal{X}$ to a set of key-value pairs over a shared key space $\mathcal{K}$, where each key is a named field (e.g., WagesAmt) and each value is the computed amount. The agent can query O on any input but cannot access its internals. A domain knowledge base $\mathcal{KB}$ (IRS form instructions, forms, XSD schemas) grounds all diagnosis. The objective is:

$$\max_C F_1^{\text{scn}}(X_{\text{train}}; C) \quad (1)$$

where F_1^{scn} is the per-scenario macro- F_1 defined in Eq. 2. Optimization observes only divergences on X_{train} , never O ’s internals or test inputs. The codebase C consists of declarative form schemas evaluated by a schema-agnostic engine; the initial schemas C_0 are LLM-bootstrapped and need not produce correct output.

Metric. We compare the replacement’s emitted (field, value) pairs against the oracle’s: a true positive is a pair both emit and agree on (currency within \$1); a false positive is a pair the replacement emits but the oracle does not; a false negative is a pair the oracle emits but the replacement does not. We measure alignment with the F_1 score over these counts, $F_1 = 2 \text{TP} / (2 \text{TP} + \text{FP} + \text{FN})$, reported in two macro views over the N scenarios X and the form set $\mathcal{F}$:

$$F_1^{\text{scn}} = \frac{1}{N} \sum_{x \in X} F_1(x), \quad F_1^{\text{form}} = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} F_1^f, \quad (2)$$

where $F_1(x)$ scores scenario x and F_1^f pools $\text{TP}_f, \text{FP}_f, \text{FN}_f$ across all scenarios for form f . *Per-scenario* is dominated by the forms in every return (Form 1040, W-2); *per-form* weights every form equally and so exposes the long tail of forms.

3 System

3.1 Tax Domain Instantiation

Oracle. A production tax preparation system representing decades of continuous development by

hundreds of engineers. Proprietary languages, hundreds of decoupled components. Queryable: feed it a taxpayer scenario, receive IRS MeF-compliant e-file XML. No source code or internals accessible.

Target. The target system separates a fixed evaluation engine from mutable form schemas. Tax forms are declarative Pydantic v2 (Colvin and Pydantic Contributors, 2024) schemas where each field carries its calculation expression and IRS XML mapping as metadata. The engine is schema-agnostic: it evaluates any conforming schema without modification. The agent reads and edits schemas, never the engine. Schemas can be LLM-bootstrapped from publicly available IRS forms, instructions, and XSD definitions. We start from C_0 , a self-consistent snapshot of the pre-agent system: the evaluation engine is mature, but the LLM-bootstrapped form schemas emit almost no correct output on these scenarios. CORAL iteratively refines these schemas toward the oracle’s behavior.

3.2 Agent Architecture

The CORAL loop has four phases (*compare*, *diagnose-and-fix*, *code-quality review*, and *verify*), supported by a domain knowledge base and a persistent defect store.

Defect store. The agent maintains a persistent defect store carried across rounds and updated in every phase: a set of canonical defects (one per root cause), each with a state and an attempt history,

$$\mathcal{D} = \{d\}, \quad \sigma: \mathcal{D} \rightarrow \Sigma, \quad h(d) = \langle \Delta F_1^{(t)} \rangle_t, \quad (3)$$

where Σ is the state set (OPEN, DIAGNOSED, FIXED, DEFERRED, WONT_FIX), $\sigma(d)$ the current state of defect d , and $h(d)$ the metric deltas ΔF_1 of its prior attempts. New mismatches from the comparison are deduplicated into $\mathcal{D}$ by root cause, so a single missing upstream form is one entry rather than dozens of downstream ones. A *deferred* defect is held for a cooldown of c rounds before it re-enters the queue, and after r retries without acceptance $\sigma(d)$ becomes WONT_FIX. This memory lets the sequencing policy revisit forms across rounds and prevents oscillation and circular repairs.

Phase 1: Compare (deterministic). Runs both systems on the same scenarios and compares e-file XML field-by-field, classifying each key-value pair as matching, mismatching, missing, or spurious, and computing per-scenario F_1 .

Phase 2: Diagnose and fix (agent + KB). Two agents operate in sequence. The *diagnosis agent* (read-only) receives comparison summaries with exact values from both systems and queries the KB (IRS instructions, forms, XSD schemas) to produce structured defect reports (root cause, affected forms, fix strategy) that populate $\mathcal{D}$. Each round targets a single below-floor form (round-robin, Eq. 5); the round’s defects are that form’s diagnosed mismatches, deduplicated by root cause so interdependent downstream ones count as a single defect, and capped at k (highest expected F_1 gain first if more remain). The *fix agent* (write access) draws the round’s selected defects from $\mathcal{D}$ with their full context (diagnosis, the attempt history $h(d)$, baselines), traces each through the data pipeline, and resolves **up to k defects per round**, after which the gate accepts or rejects the round as a unit, bounding the blast radius. It runs comparisons on its own and iterates until confident the fix will clear the gate; all existing tests must pass and new code must add passing tests.

Phase 3: Code quality review (agent + standards). Because the gate scores outputs alone, an F_1 -optimizing agent will take any shortcut that passes (a hardcoded constant, a one-off suppression rule, copy-pasted logic), trading maintainability for score. A review sub-agent counters this: before a fix is committed, it diffs the change against a coding-standards document, refactors away the shortcuts, and re-runs the tests and comparison to confirm behavior is unchanged. The review shapes code quality, not acceptance (the deterministic gate of Phase 4 decides that), so a fix it cannot clean without risking correctness is still committed, tagged as quality debt.

Phase 4: Verify (deterministic gate). The gate re-executes *all* training scenarios and recomputes the per-scenario F_1 , written $F_1^{\text{scn}}(C)$ for codebase C . For the defect $d \in \mathcal{D}$ under repair, a candidate edit $C \rightarrow C'$ with $\Delta F_1 = F_1^{\text{scn}}(C') - F_1^{\text{scn}}(C)$, tolerance $\delta = 0.05$, and T the event that all tests pass, it updates the codebase C and the defect store $\mathcal{D}$:

$$(C, \mathcal{D}) \leftarrow \begin{cases} (C', \mathcal{D} \setminus \{d\}) & \Delta F_1 > 0, \\ (C', \text{defer}(d)) & \Delta F_1 \in [-\delta, 0], \\ (C, \mathcal{D}) & \text{otherwise,} \end{cases} \quad (4)$$

where $\text{defer}(d)$ denotes $\mathcal{D}$ with d ’s state set to DEFERRED, and the first two cases require T . A pos-

itive ΔF_1 marks d fixed and removes it from the store; a change within the tolerance band keeps the edit and defers d (cooldown and escalation as above); the *otherwise* case (a failed test or $\Delta F_1 < -\delta$) rolls back, leaving C and $\mathcal{D}$ unchanged so d is retried. Deferral is a form of acceptance that keeps the edit; only $\Delta F_1 < -\delta$ or a failed test discards one. Acceptance is decided by rerunning the training scenarios as a batch-level regression check, not by model self-assessment, the key distinction from self-debugging.

Sequencing. The loop chooses which form to repair next by round-robin over under-covered forms, not by whichever defect most raises the per-scenario score. Let $F_1^f(C)$ be the F_1 of form f (§2) and τ a coverage floor. Each pass visits every form in the below-floor set

$$\mathcal{B}(C) = \{f \in \mathcal{F} : F_1^f(C) < \tau\} \quad (5)$$

once and repeats until $\mathcal{B}$ converges, i.e., no form in it admits further meaningful improvement (the remainder having escalated as data gaps); it need not empty. A form leaves $\mathcal{B}$ when $F_1^f \geq \tau$ or is escalated; otherwise it is revisited on a later pass, such that a fix blocked on another form is retried once that form is repaired. This spreads effort across the full form distribution rather than the high-frequency head, which matters because a return is correct only if *every* form it requires is correct: a long-tail form is as essential to the taxpayer who needs it as Form 1040 is to everyone.

Escalation. The agent escalates a defect to humans, setting $\sigma(d) = \text{WONT_FIX}$ in $\mathcal{D}$, when: the governing IRS rule is ambiguous (more than one defensible treatment), input data is incomplete (not a code defect), it exhausts the r cooldown retries above, or the fix requires architectural changes beyond the agent’s current tool scope.

3.3 Data

We use 119 taxpayer scenarios synthetically curated by tax-analyst subject-matter experts (SMEs) to cover filing situations; because they are synthetic, they contain no real taxpayer data or personally identifiable information (PII). For each, the oracle produces an e-file XML return as ground truth.

Source documents and scenario variation. Each scenario is a complete taxpayer return assembled from a small set of input documents, chiefly

Output form	n	Output form	n
Form 1040	119	Schedule C	7
W-2	69	Schedule 8812	6
Schedule B	38	Form 8814	6
Form 8959	28	OtherIncome stmt.	6
Schedule 2	19	Schedule 3	5
Form 8960	18	ChildInterest stmt.	4
Schedule D	16	Schedule A	4
Schedule 1	14	Form 172	2
Form 8867	14	Form 4137	1
Payment record	12	Form 8949	1
Form 8995	10	Form 8995-A	1
Form 2441	9	OtherWithhold. stmt.	1

Table 1: The 24 output forms with the number of scenarios including each (n , of 119). The form-stratified 59/30/30 split places every form (singletons included) in train, so train covers all 24; per-form macro F_1 weights this long tail equally. Form 8880 is *not* among these corpus forms: no scenario requires it, but the replacement over-emits it, so it appears at $F_1=0$ in Figure 3.

a W-2 (wages), a 1099-INT (interest), and a 1099-DIV (dividends), with nearly half of the returns combining two or more. Beyond the document mix, taxpayer-specific choices, such as filing status (e.g., MFJ vs. single), dependents, and self-employment, among others, change a return’s complexity by determining which forms must be filed, and hence which forms appear in the output.

Output forms. Every return centers on Form 1040 and its schedules, with situational forms (e.g., 8959, 8960, 8995, 2441, 8867, 8814) and supporting statements. Across the pool, returns span 24 distinct forms (Table 1), a single scenario emitting from a handful to more than ten with cascading cross-form dependencies. This long tail of forms appearing in only a few returns is what per-form macro F_1 weights equally.

Split. 59 train / 30 val / 30 test, stratified for *coverage* (every test form also appears in a train and a validation scenario) but not *frequency*. The lighter validation split (3.0 vs. 4.2 forms per return) gives noisy per-form estimates, so we use it only for checkpoint selection and report train and test, which contain the full form tail. The agent observes only the train split.

4 Experiments

We study three questions about aligning a system to a black-box oracle from outputs alone:

Q1 Feasibility and generalization. How closely can an agent align the replacement to the ora-

Configuration	F_1^{scn}	F_1^{form}
C_0 (no agent)	0.169	0.032
Open-loop (no feedback)	0.852	0.496
CORAL (gated)	0.867	0.321
CORAL (gated+RR)	0.905	0.704

Table 2: Test-split macro- F_1 per configuration at its best-validation checkpoint (RR: round-robin); F_1^{scn} is per-scenario and F_1^{form} per-form (§2). All start from C_0 and use the same evaluator.

cle from feedback alone, and do its fixes generalize to unseen inputs?

Q2 Verification. Is a deterministic accept/reject gate necessary for reliable improvement, or can the model self-assess?

Q3 Defect sequencing. How does the policy for choosing which defect to repair next shape the optimization trajectory and the alignment the agent ultimately reaches?

We ablate the loop’s two components from a common start C_0 : the *open-loop* agent has neither gate nor sequencing; **CORAL (gated)** adds the verification gate (Eq. 4); and **CORAL (gated + round-robin)** adds round-robin sequencing (Eq. 5). Open-loop vs. gated isolates verification (Q2), the two CORAL variants isolate sequencing (Q3), and the full system’s train–test gap measures generalization (Q1).

Setup. Every configuration starts from the same C_0 , a mature engine with LLM-bootstrapped schemas at macro F_1 of only 0.17 per-scenario and 0.03 per-form (Table 2). All see the same 59 train / 30 val / 30 test scenarios, are scored by the same canonical evaluator, with per-round batch size $k=5$ and coverage floor $\tau=0.95$ (orchestration and model in Appendix A). We report the per-scenario and per-form macro- F_1 of §2; per-form is the demanding view, since a taxpayer needs every form their return requires. Checkpoints are selected by best validation per-scenario F_1 .

4.1 Verification is necessary

To test whether a verification gate is necessary (Q2), we run an **open-loop** agent that commits every edit. Its trajectory slips below its peak (Figure 2, left), ending at test per-scenario 0.852 and per-form 0.496 (Table 2). The deterministic gate (Eq. 4) rolls back any round that lowers the train per-scenario F_1 beyond a small tolerance, so error does not

accumulate and the two gated runs hold their best checkpoint: verification is necessary (Q2).

Open-loop edges the gated-only run on per-form F_1 (0.496 vs. 0.321), but this reflects how each is steered, not a better repair process. With no feedback, open-loop works through every surfaced defect and so touches the whole form distribution, tail included; the gated-only run steers by the head-dominated per-scenario F_1 (Form 1040, W-2) and neglects the long tail (§4.2), depressing its per-form score. Open-loop buys that breadth with instability. The gate’s role is to stop error from accumulating (Q2), not to secure coverage; that is what form-level sequencing adds. The full system, pairing both, surpasses open-loop on both views (per-form 0.704, per-scenario 0.905). Both CORAL configurations use the gate and differ only in how they pick the next form.

4.2 Gated abandons the tail

CORAL (gated) isolates how the objective shapes coverage (Q3): it uses the gate but no form-level sequencing, so its only steering signal is the per-scenario F_1^{scn} . Because F_1^{scn} is dominated by the head forms, effort flows there: it raises Form 1040 and W-2 to near-1.0 F_1 (hill-climbing that score) while the long tail of forms stays unimproved. The left panel of Figure 3 makes this visible: the tail rows stay grey or red across *every* checkpoint (six never emitted, four emitted but never corrected), so the gated agent never works on them. This is a property of the objective, not a capability limit: a form that barely moves F_1^{scn} gives the agent almost no reason to visit it.

The same objective also transfers worse to held-out data: the gated agent’s per-scenario score falls from 0.950 on train to 0.867 on test (Figure 2), a wider train–test gap than the full system shows. On test it trails the full system on both viewpoints (per-scenario 0.867 against 0.905, per-form 0.321 against 0.704); it leads only on the training per-scenario score it optimizes (0.950 against 0.922).

Repeated revisits and premature stopping. With only the per-scenario score to steer by, the gated agent cycles on the head: 24 of its 92 commits (26%) touch Schedule B, each adding a suppression rule, drawing a regression, and reverting, without converging. It declines tail work, dismissing the two scenarios that need a not-yet-built Form 8814 schema as “*too complex to fix*” and terminating with “*accept that the train F_1 is*

plateaued.” These forms remain at $F_1 = 0$: a per-scenario objective cannot direct effort to under-weighted forms, leaving the tail unserved (Q3).

4.3 Round-robin recovers the tail

CORAL (gated + round-robin) keeps the gate and adds round-robin sequencing over under-covered forms (Eq. 5): each pass gives every below-floor form one turn, forcing the agent onto a different form each round rather than the head forms that dominate the per-scenario score (Q1, Q3). The right panel of Figure 3 shows the effect: the tail forms recover one after another instead of staying dead. Schedules 2 and 3 rise from 0.4 to 1.0, Form 8960 from 0.43 to 0.84, and ChildTaxableInterestStmt from 0 to 0.50. Coverage doubles, from 7 to 14 forms at $F_1 \geq 0.5$; Schedule C, dead under the gated agent, recovers to 0.86. Only two forms remain at $F_1 = 0$: Schedule A, whose triggering data is absent, and Form 8880, which the replacement over-emits in a boundary case the published rules do not clearly settle; both are flagged for manual audit rather than left as silent errors.

Forcing a fresh form each round also keeps the agent from getting stuck: where the gated agent cycles on Schedule B, round-robin keeps surfacing new defects on new forms, so it attempts more genuine repairs and reaches a higher per-scenario score too. It leads on *both* test viewpoints, per-scenario **0.905** (against 0.867) and per-form **0.704** (against 0.321). The per-form number is a coverage measure, not a per-return error rate: averaging every form equally, it is dominated by the long tail, whereas the common forms (Form 1040, W-2, and the major schedules) are matched far more closely. These gains generalize (Q1): train and test agree closely (per-scenario 0.922 vs. 0.905; per-form 0.77 vs. 0.70), a much smaller train–test gap than the gated agent’s, indicating genuine repair rather than memorization. Because the split is stratified so every test form also appears in training, this measures generalization to unseen taxpayer scenarios composed of previously seen forms, not to entirely unseen forms or new tax rules. The gate alone is not enough: on its own the agent perfects the head and abandons the tail. Verification makes progress reliable (Q2); the sequencing policy decides which alignment the agent reaches (Q3).

4.4 Code quality subagent

The gate rewards output agreement but says nothing about how the code reads, so an F_1 -optimizing

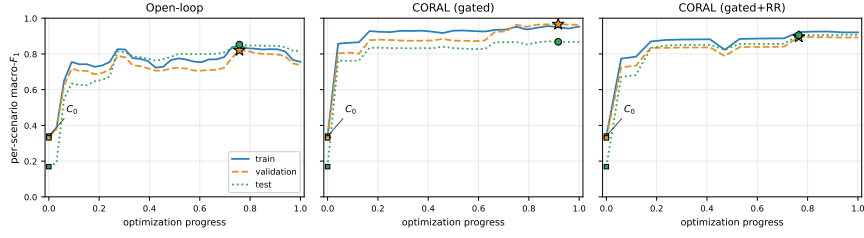


Figure 2: Per-scenario macro- F_1 (train/val/test) over each run, progress normalized to $[0, 1]$; stars mark the validation-selected checkpoint, circles its test score. Open-loop deteriorates below its own peak; the gated runs follow a smoother path and hold their best. The one dip in the gated + round-robin curve is a round admitted within the 0.05 tolerance (Eq. 4) and corrected the next round.

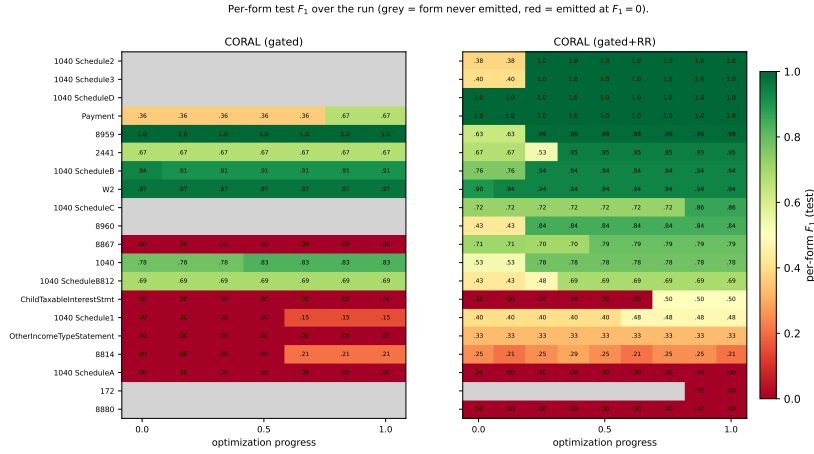


Figure 3: Per-form test F_1 for each form (rows, sorted by the gated + round-robin agent’s final score) over the run (columns). Eighteen rows are corpus forms (Table 1) present in the held-out test split; the nineteenth, Form 8880, is emitted by the replacement on test scenarios though no scenario requires it (a false-positive form, scored at $F_1=0$). The remaining 6 of the 24 corpus forms occur only in train/validation scenarios and have no test score to plot. Grey marks a form never emitted; red marks a form emitted at $F_1=0$. Left: gated. Right: gated + round-robin. Under gated, 10 of 19 forms stay dead; with round-robin only 2 remain, the rest rising round by round.

agent can pass with brittle shortcuts (a broad Exception catch, a hardcoded constant) that leave outputs unchanged and so escape the gate. The Phase-3 review targets this: it runs on each submitted fix before the deterministic regression gate (Phase 4), and refactors the fix in 67% of rounds (shared helpers, new tests, corrected XML keys). The gate secures correctness; an explicit standard secures maintainability.

4.5 Defect Taxonomy

Table 3 breaks the 66 defects of the gated + round-robin run into categories and shows where round-robin spends its effort. Trigger defects dominate (24 fixed): for example, Schedule B failing to emit for interest and dividend income. Reviving a dead form is first a matter of making it emit, and coverage sequencing front-loads exactly these. Data-flow defects come next (16), such as a W-2 state code left unresolved: once a form emits, its fields

must be populated from the taxpayer’s source documents, which is where most of the remaining work lies. Calculation (6; e.g., a foreign-tax-credit limit ignored) and XML-serialization (5; e.g., a missing XML annotation) defects are fewer. All are schema-level; because the engine is mature, none is an engine or pipeline defect. The escalated column (13) previews the capability boundary, which we examine next.

4.6 Capability boundary

The 13 escalated defects mark the agent’s capability boundary (Table 4) and explain most of the residual per-form gap: the 0.70 score falls short because these forms hinge on missing inputs or ambiguous IRS rules, not errors in the forms the agent repaired. A tax professional confirmed all 13 reflect missing inputs, ambiguous rules, or low-yield residuals. Some are *scenario data gaps*: the target depends on an input the scenario lacks, such as

Category	Fixed	Esc.
Form trigger	24	3
Data flow	16	4
Calculation logic	6	1
XML serialization	5	1
Trigger suppression	2	3
Schema completion	0	1
Total	53	13

Table 3: The 66 defects of the gated + round-robin run by category (53 fixed, 13 escalated).

Escalation reason	<i>n</i>
Scenario data gap	5
Ambiguous IRS rule	3
Deferred residual	5
Total	13

Table 4: The 13 escalated defects by reason.

filing status, while the oracle may apply a default setting and still emit a correct output. In such cases, the agent traces the mismatch to the missing input rather than inventing a value.

A second group turns on *ambiguous IRS rules*: the regulation admits more than one defensible treatment, so committing to one risks false positives elsewhere (e.g., whether to serialize a zero-valued tax field under MeF conventions, or suppress Form 8880 in an edge case). The agent escalates these for a tax professional to review. The remaining cases are *deferred residuals*: partial fixes with small remaining error, or defects escalated after repeated low-yield attempts.

5 Related Work

Coding agents. SWE-bench (Jimenez et al., 2024), SWE-agent (Yang et al., 2024), RepairAgent (Bouzenia et al., 2025), and OpenHands (Wang et al., 2025) resolve *pre-specified* issues in accessible repositories. CORAL instead *discovers* defects from oracle/replacement disagreement and carries them through a persistent defect lifecycle over many rounds.

Test-based automated repair. GenProg (Weimer et al., 2009) and SemFix (Nguyen et al., 2013) optimize against developer-specified tests, a sparse pass/fail signal that already encodes the spec. CORAL optimizes against dense field-level disagreement from a production oracle, under a deterministic gate with rollback/defer policies.

Self-improvement loops. Reflexion (Shinn et al., 2023), Self-Debugging (Chen et al., 2024), CodeRL (Le et al., 2022), and AgentCoder (Huang et al., 2023) improve code through model-mediated critique or test outcomes. CORAL externalizes supervision: acceptance is a deterministic metric change, not model self-assessment.

Migration and transformation. LLMLift (Bhatta et al., 2024) and industrial migration at Google (Ziftci et al., 2025) do source-to-source transformation but assume readable codebases on both sides. Our setting is harder: alignment to a *black-box*, query-only legacy system.

Oracle-guided synthesis and selection. Oracle-guided synthesis (Jha et al., 2010) and candidate selection (Fan et al., 2024) use I/O signals for small programs or ranking. CORAL scales oracle guidance to a real, interdependent codebase.

Tax-domain context. TaxCalcBench (Bock et al., 2025) shows the gap between frontier models and production-grade tax correctness. CORAL treats LLMs as repair agents around a deterministic engine, using IRS documentation and oracle comparison rather than end-to-end generation.

6 Conclusion

We introduced CORAL, a framework for *oracle-only modernization*: it aligns a modern U.S. individual income tax engine (Form 1040) to a black-box legacy oracle using only the oracle’s input-output behavior and authoritative IRS documentation, so every repair traces to published regulation rather than model knowledge. A deterministic F_1 gate verifies each edit by re-execution and a persistent defect store prevents oscillation, while round-robin sequencing over under-covered forms drives repair into the long tail a per-scenario objective would otherwise abandon. Across a corpus of 119 tax-expert-authored scenarios it lifts per-scenario macro- F_1 on a 30-scenario held-out test split from 0.17 to 0.90 and yields a taxonomy of 53 fixed and 13 escalated defects, the latter routing ambiguous rules and input-data gaps to manual audit. Future work includes synthetic input generation from the queryable oracle, active scenario selection, and transfer to other legacy systems used as oracles.

Limitations

1. **Single domain.** Validated only on US federal tax (Form 1040 family). Generalizability to other structured-output domains is plausible but not demonstrated.
2. **LLM-bootstrapped start.** The target started from LLM-bootstrapped schemas ($C_0 \neq \emptyset$). The framework does not address cold-start from an empty codebase.
3. **Single model.** All experiments use a single model, Claude Opus 4.6. Comparison across model families and providers is future work.
4. **Single trajectory per configuration.** Each configuration is one long-horizon agent run; we do not estimate run-to-run variance.
5. **No public release.** The oracle, scenario corpus, and codebase are proprietary; contractual constraints prevent releasing the system or the paired scenario/e-file data.

Ethics Statement

Authorization and ownership. The production system used as the oracle is owned and operated by the authors’ organization, and this work was carried out with the organization’s permission.

Data and privacy. The 119 evaluation scenarios were synthetically curated by tax-analyst subject-matter experts (SMEs) to exercise specific filing situations. Because they are synthetic, they contain no real taxpayer data or personally identifiable information (PII); no real taxpayer return was processed at any point in this study.

Authority and human oversight. The oracle engine is the alignment target because it encodes decades of applied IRS regulation. However, every diagnosis is grounded in IRS documentation. The two rarely diverge; where they appear to (e.g., an ambiguous rule), CORAL escalates the case to a tax professional for resolution, so no non-compliant behavior is silently encoded. CORAL is a developer aid for building and maintaining a regulated engine under human supervision, not an autonomous tax-filing agent. All outputs are intended for review by tax professionals before any real-world use.

References

- Anthropic. 2025. [Claude agent SDK](#).
- Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit Seshia, and Alvin Cheung. 2024. Verified code transpilation with llms. *Advances in Neural Information Processing Systems*, 37:41394–41424.
- Michael R Bock, Kara Molisee, Zachary Ozer, and Sumit Shah. 2025. Taxcalcbench: Evaluating frontier models on the tax calculation task. *arXiv preprint arXiv:2507.16126*.
- Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. Repairagent: An autonomous, llm-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2188–2200. IEEE.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pages 8746–8825.
- Samuel Colvin and Pydantic Contributors. 2024. [Pydantic: Data validation using Python type hints](#).
- Zhiyu Fan, Haifeng Ruan, Sergey Mechtaev, and Abhik Roychoudhury. 2024. Oracle-guided program selection from large language models. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 628–640.
- Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agent-coder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.
- Internal Revenue Service. 2024a. [About form 1040, U.S. individual income tax return](#).
- Internal Revenue Service. 2024b. [Modernized e-file \(MeF\) program information](#).
- Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.
- Ravi Khadka, Belfrit V Batlajery, Amir M Saeidi, Slinger Jansen, and Jurriaan Hage. 2014. How do professionals perceive legacy systems and software modernization? In *Proceedings of the 36th International Conference on Software Engineering*, pages 36–47.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE.

Robert C Seacord, Daniel Plakosh, and Grace A Lewis. 2003. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Wannita Takerngsaksiri, Jirat Pasuksmit, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Ruixiong Zhang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. 2025. Human-in-the-loop software development agents. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 342–352. IEEE.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. Llm4decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3473–3487.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2025. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pages 65882–65919.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374. IEEE.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Dececasa, and Max Kim. 2025. Migrating code at scale with llms at google. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 162–173.

A Implementation

We implement CORAL with the Claude Agent SDK (Anthropic, 2025), running Claude Opus 4.6 at high thinking effort. A single *orchestrator* drives the loop and owns all shared state (the codebase

C , the defect store $\mathcal{D}$, and the current metrics). It runs the two deterministic phases itself, Phase 1 (execute both systems, compute the field-level comparison and per-scenario F_1) and Phase 4 (the gate of Eq. 4: re-run the training scenarios and the unit-test suite, then apply the codebase and defect-store update), and it spawns a fresh *sub-agent* for each agentic phase. In the SDK’s terms, a *sub-agent* is a separate Claude instance the orchestrator delegates a phase to, with its own context window, system prompt, tool set, and permission scope; it runs to completion and returns only a structured result, isolating each phase’s context, which the orchestrator records in $\mathcal{D}$ before the next phase.

Tools are exposed through a Model Context Protocol (MCP) server whose operations cover defect-store reads and writes, batch scenario execution against the oracle, field-level output comparison, and read-only knowledge-base queries. The orchestrator grants each sub-agent only the operations its role requires:

- **Diagnosis** (read-only): file read and search, scenario comparison, and KB queries, but no write tools. Withholding edits forces it to establish and cite a root cause before any change is attempted, and keeps diagnosis grounded in the IRS knowledge base rather than tied to one candidate patch.
- **Fix** (read, plus write to schemas): the read-only tools, schema editing, and the unit-test runner. It may edit form schemas but not the evaluation engine, enforcing the schema/engine separation of §3.1; every change must add passing tests before it reaches the gate.
- **Code-quality review** (read, plus write to schemas): the same scope as the fix sub-agent, used only to refactor an already-accepted fix against a coding-standards document and re-run the tests and comparison to confirm behavior is unchanged.

The orchestrator enforces these scopes, selects and batches up to k defects per round, applies the gate to the round as a unit, and manages the cooldown and escalation transitions over $\mathcal{D}$. Figure 4 summarizes the loop.

B Reproducibility and Configuration

All configurations share the setup in Table 5 and the orchestration and tool scopes of Appendix A. The

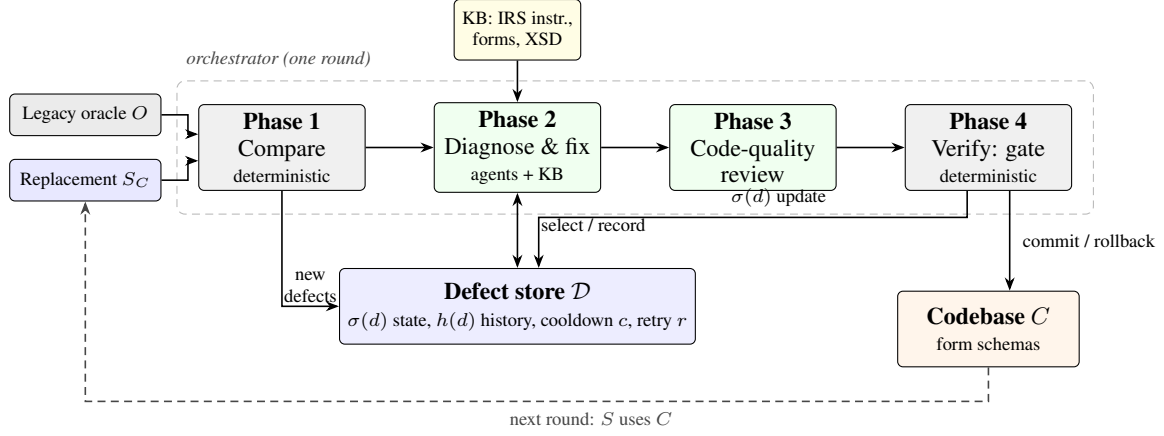


Figure 4: The CORAL loop for one round. The orchestrator runs the deterministic Phases 1 and 4 and spawns a sub-agent for each of Phases 2 and 3. Both systems feed the field-level comparison; the knowledge base grounds diagnosis. The persistent defect store $\mathcal{D}$ is read and written across phases (new defects in Phase 1, selection and recording in Phase 2, the $\sigma(d)$ update in Phase 4). The gate then *commits* the edit ($C \leftarrow C'$) or *rolls back* (keeps C); the committed codebase parameterizes the replacement S_C for the next round.

loop runs until the training per-scenario F_1 plateaus (equivalently, until the below-floor set $\mathcal{B}$ of Eq. 5 admits no further improvement); we then select the checkpoint by best validation per-scenario F_1 .

C Sub-agent Prompts

Setting	Value
Model	Claude Opus 4.6 (high effort)
Batch size k	5 defects / round
Coverage floor τ	0.95
Gate tolerance	$\Delta F_1 \geq -0.05$, tests pass
Currency tol. ϵ	\$1
Escalation	3 consecutive failed tries
Cooldown c	15 rounds
Checkpoint	best validation per-scenario F_1
Knowledge base	IRS instructions, forms, XSDs

Table 5: Shared configuration. Acceptance follows Eq. 4: a within-tolerance regression is admitted only when every test passes; a larger drop or a failed test is always rolled back. (The fix agent’s own iteration before submission, Phase 2, is separate from this deterministic gate.)

Bootstrapping C_0 . The initial schema collection C_0 was generated in a single pass by an LLM agent given read access to the IRS knowledge base. Prompted to consult each form’s instructions, form, and MeF XSD schema, the agent emitted the corresponding Pydantic form schemas (fields, calculation expressions, and XML annotations) for the Form 1040 family in one shot, with no oracle comparison or iterative refinement. This yields a non-empty but weak starting point (macro- F_1 of 0.17 per-scenario and 0.03 per-form) that CORAL then repairs.

The prompts below are illustrative; each sub-agent also has the role-scoped tool set of Appendix A and the IRS knowledge base.

Diagnosis sub-agent (read-only)

- Input: per-field comparison for the affected scenarios, with the oracle’s and replacement’s exact values.
- Query the IRS knowledge base (form instructions, forms, MeF XSD schemas) to explain *why* the values differ, and cite the governing IRS rule.
- Output a structured defect report: root cause, affected forms and fields, and a proposed fix strategy grounded in a cited rule.
- Do not edit code; deduplicate mismatches that share one root cause into a single canonical defect.

Fix sub-agent (write access)

- Input: the round’s selected defects with diagnosis, prior attempts and their metrics, and current baselines.
- Ground the change in the diagnosis’s cited IRS rule; trace each defect through the data pipeline before editing; change *schemas only*, never the engine.
- Add unit tests for new logic; all existing tests must pass.
- Re-run comparisons yourself and iterate until confident the change will clear the gate.

Review sub-agent (coding standard)

- Input: the fix diff and the coding-standards document.
- Refactor shortcuts (duplicated logic, broad except, private-attribute access, wrong XML keys) without altering behavior.
- Re-run tests and comparisons to confirm outputs are unchanged; if a clean-up risks correctness, commit as-is and tag it as quality debt.

D Worked Repairs

Three illustrative repairs; each shows the diagnosis, the schema edit, and the gate outcome.

Calculation, KB-grounded (accepted). The replacement leaves the Additional Medicare Tax at zero. The diagnosis sub-agent, citing the Form 8959 instructions (0.9% on Medicare wages above a filing-status threshold), replaces the stub with the computation:

form_8959.py: Additional Medicare Tax

```
- line_7 = 0.0 # not computed
+ line_7 = Calc(lambda ctx: round(
+     max(0, ctx["med_wages"]
+     - THRESH[ctx.status]) * 0.009))
```

The gate re-executes all training scenarios, $\Delta F_1 > 0$, and accepts.

Form trigger, reviving a dead form (accepted). Schedule B never emits, so all its fields read as missing. The diagnosis is a trigger gap: interest or dividends above \$1,500 require the schedule, but the predicate is stubbed off.

schedule_b.py: trigger predicate

```
- return False # never emits
+ return (interest_total > 1500
+         or dividend_total > 1500)
```

The form now emits; its per-form F_1 rises from 0, and the gate accepts.

Scenario data gap (escalated). The return is Married Filing Jointly, but the extracted scenario carries no spouse identity, so the replacement reads Single and downstream brackets diverge. The agent traces the mismatch to a missing input rather than a code defect and escalates instead of inventing a filing status.

E Schema Anatomy

Figure 5 shows a single field of a form schema. Everything the engine and the agent need is local to the field:

- `Calc(...)`: the deterministic calculation, evaluated by the schema-agnostic engine.
- `description, irs_reference`: self-documentation the diagnosis sub-agent reads to ground a fix.
- `data_sources`: provenance as an immutable (source, path) tuple, enabling the audit trail.
- `xml_annotation`: the IRS MeF tag and type used at serialization.

This locality is what makes a repair a small, reviewable schema edit rather than a change to black-box engine logic.

F Defect Catalog (Representative)

Table 6 lists representative defects behind the aggregate counts of Tables 3 and 4, illustrating each category with a concrete root cause and disposition.

form_1040.py: anatomy of one schema field

```

line_12a: float = Field(
    default=Calc(lambda ctx: STD_DEDUCTION[ctx.filing_status]),
    description="Standard deduction for the filing status;
    see the IRS Form 1040 instructions.",
    json_schema_extra={
        "field_type": FieldType.CURRENCY,
        "data_sources": [("taxpayer_profile", "data.filing_status.value")],
        "xml_annotation": XmlAnnotation(tag="TotalDeductionsAmt",
            type="USAmountType", use="required"),
        "irs_reference": "Form 1040, Line 12a",
    },
)

```

Figure 5: Anatomy of one self-documenting schema field (illustrative). The calculation lives with the field as a Calc expression; description and irs_reference make it human- and LLM-readable; data_sources declares provenance as a (source, path) tuple; and xml_annotation carries the exact MeF mapping. The agent edits fields like this one; the engine that evaluates them is never touched.

Category	Example defect (root cause)	Fix strategy	Disp.
Form trigger	Schedule B not emitted for interest/dividends	add trigger predicate	Fixed
Form trigger	Form 8959 not emitted above Medicare threshold	add trigger predicate	Fixed
Data flow	W-2 state code (box 15) left unresolved	map source field	Fixed
Data flow	1099-DIV qualified-dividend amount unmapped	map source field	Fixed
Calculation logic	Foreign-tax-credit limit ignored	add limit calculation	Fixed
XML serialization	Missing XML annotation on a credit field	add xml_annotation	Fixed
Trigger suppression	Schedule B over-emitted on a no-interest return	add suppression predicate	Fixed
Trigger suppression	Replacement over-emits Form 8880 (unsettled edge case)	suppression rule ambiguous	Esc.
Data flow	Filing status read as Single; should be MFJ	input missing from scenario	Esc.
Schema completion	Zero-valued tax/withholding field serialization	MeF convention ambiguous	Esc.

Table 6: Representative defects by category from the 66-defect run. Trigger and data-flow defects are localized, single-file, and groundable in the knowledge base; escalated rows reflect missing inputs or ambiguous tax rules, not diagnosis failures.