Compiler Compiler Technology: Round-Trip Testing as an Oracle

Definition, Properties, and Application in CCT Methodology

A. F. Urakhchin

Draft 0.1 — May 2026

Defines round-trip testing as an oracle for validating information-preserving transformations, characterizes the conditions under which it provides structural correctness evidence, and describes its application in CCT methodology as the primary oracle for AP-generated code.

Companion documents:

CCT: Operational Disciplines for Adaptive Programming (Draft 0.1)

CCT: Adaptive Programming with Claude Code (Draft 0.1)

1. Purpose and Scope

1.1 What This Document Is

This document defines round-trip testing as an oracle for validating information-preserving transformations, characterizes the conditions under which it provides structural correctness evidence, and describes its application in CCT methodology as the primary oracle for code generated through Adaptive Programming. The document is intended as a standalone reference: other CCT documents (the Operational Disciplines, the Adaptive Programming guide, the MCP Server roadmaps) refer to round-trip testing as an oracle without defining it; this document is what those references point to.

The treatment is general where the concept is general and specific where the application is specific. Chapters 2 through 4 define round-trip testing in terms applicable beyond CCT — to any domain involving information-preserving transformations between representations. Chapter 5 describes how the CCT substrate enables round-trip testing at scale. Chapters 6 through 8 cover applicability conditions, comparison to other validation approaches, and practical construction patterns.

1.2 Why Round-Trip Testing Deserves Standalone Treatment

Round-trip testing has unusual properties as a validation approach. Most oracles compare a program's output against an expected value supplied by the test author — the test author must know what the right answer is, and the test is only as good as the author's knowledge. Round-trip testing instead validates that two transformations are consistent inverses of each other, without requiring the test author to know what either transformation's intermediate output should be. This makes round-trip testing applicable to cases where expected outputs would be impractical to author — large grammars, complex data models, mechanically-generated structures — and shifts the validation burden from "is the output correct" to "is the pair of transformations consistent."

In CCT specifically, round-trip testing is load-bearing. It is the oracle that validates SGDL grammar definitions are correctly compiled to parsers (the parsed output of a grammar's own source should round-trip through that parser); it is the oracle that validates the cct_repo persistence layer (textual .cctrepo content should round-trip through the SQLite-backed database with byte-identical fidelity); it is the oracle that validates AP-generated code (transformations the generated code implements should preserve information end-to-end). Round-trip testing is the reason large amounts of code can be generated and trusted without manual verification of each generated line.

1.3 Audience and Prerequisites

The document assumes familiarity with software testing concepts at the level of unit testing and integration testing. No background in formal methods or category theory is required, though readers familiar with those frameworks will recognize round-trip testing as an instance of natural transformations or adjoint functor pairs, depending on the formalism. The CCT-specific chapters assume familiarity with CCT terminology — SGDL grammar, SCB (Syntax-Controlled Binary), cppcc — at the level of the Operational Disciplines document.

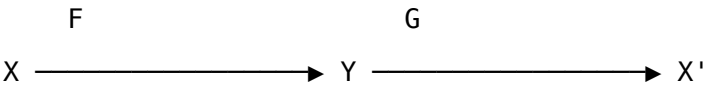
1.4 Document Structure

Chapter 2 defines the round-trip pattern formally and introduces the diagrammatic conventions used throughout. Chapter 3 makes the structural argument for why round-trip testing functions as an oracle and characterizes what it does and does not validate. Chapter 4 distinguishes levels of round-trip closure — exact, canonical, semantic — and the trade-offs among them. Chapter 5 describes how the CCT substrate provides round-trip closure as a foundational service and presents specific CCT examples. Chapter 6 names the applicability conditions and limitations. Chapter 7 compares round-trip testing to other validation approaches and clarifies what it complements rather than replaces. Chapter 8 covers practical construction.

2. The Round-Trip Pattern

2.1 Formal Definition

A round-trip test consists of two transformations and a comparison. Given a representation X, a forward transformation F produces a different representation Y, and a reverse transformation G produces a representation X' that is meant to be equivalent to the original X. The round-trip succeeds if X' equals X under whatever equality relation is appropriate for the domain.




```
Test succeeds iff X' = X (under chosen equality)
```

The structure is symmetric in principle — one can also test the reverse direction by starting with a Y, transforming to X via G, then back to Y' via F, and comparing Y' to Y. In practice, one direction is usually chosen as the canonical starting point based on which representation is more naturally specified or more reliably generated; the other representation is the intermediate.

The minimal requirements for the pattern to apply are: F and G must exist (both transformations are implementable); X and Y must be comparable under defined equality relations; the transformations must be intended to be information-preserving in at least one direction. Round-trip testing is not applicable when F or G is intentionally lossy (compression that discards information, summarization, anonymization).

2.2 Closure

The pattern requires what is called closure: the composition of G after F (written $G \circ F$) should equal the identity function on the domain of X, at least restricted to the inputs the test exercises. If $G \circ F$ is identity, then any X' that comes back equal to its corresponding X is evidence that F and G are correctly implementing their intended inverse relationship for that input.

Closure is a property of the pair (F, G), not of either transformation alone. F could be correctly implemented but G could be wrong in a way that exactly cancels F's correctness — and the round-trip would still succeed. This compensating-error case is genuinely possible and is the chief limitation of round-trip testing as an oracle (Chapter 3 examines this in detail). In practice, compensating errors are rare because F and G are usually implemented independently by different mechanisms or against different specifications, and the probability of two independent errors exactly canceling under a wide range of inputs is low. Round-trip testing does not eliminate this risk but reduces it sharply through input diversity.

2.3 Variants of the Pattern

Three variants of the round-trip pattern appear regularly. The basic two-hop round-trip — $X \to Y \to X'$ — is the most common and is what the rest of this document means by "round-trip" without qualification. The

multi-hop round-trip chains additional intermediate transformations — X → Y → Z → Y' → X' — which is useful when the transformation pipeline naturally involves more than two stages (parse, transform, optimize, serialize). The bidirectional round-trip exercises both directions starting from both sides — X → Y → X' compared to X, and Y → X → Y' compared to Y. The bidirectional variant provides stronger evidence because compensating errors that survive one direction may not survive both.

2.4 The Equality Relation

The equality relation under which X' is compared to X is part of the test's specification, not an afterthought. Different equality relations produce different round-trip tests against the same transformation pair. Exact equality (X' is bit-identical to X) is the strictest and most informative; canonical equality (X' equals X after both are normalized) accommodates representations that have multiple equally-valid forms; semantic equality (X' and X have the same meaning under some semantic interpretation) accommodates representations that have many forms with the same meaning. Chapter 4 examines these levels in detail.

3. Round-Trip Testing as Oracle

3.1 What an Oracle Is

In software testing, an oracle is a mechanism for determining whether a test outcome is correct. Every test has at least an implicit oracle — the assertion that some condition holds. The strength of a test depends on the strength of its oracle: a test whose oracle is hand-coded expected output is only as good as the test author's judgment about that output; a test whose oracle is a property the output must satisfy is as good as the property's adequacy as a correctness condition; a test whose oracle is an independent reference implementation is as good as the agreement between the reference and the system under test.

Oracles vary along several dimensions. Specificity: does the oracle check one specific value ("output equals 42") or a property over many values ("output is even")? Independence: is the oracle derived from the same specification as the system under test, or independent of it? Coverage: how much of the system's behavior does the oracle validate per test invocation? Cost: how expensive is the oracle to write, to run, and to maintain?

3.2 What Makes Round-Trip Testing an Oracle

Round-trip testing is an oracle because the comparison X' = X is a correctness condition the system under test (the pair F and G) must satisfy if it is correctly implementing its intended behavior. The oracle does not require the test author to know what Y looks like, what intermediate computations happen, or what the right output is in absolute terms. The oracle requires only that the test author know what the input X is and can compare X' to it under the chosen equality relation.

The structural argument for round-trip as oracle is straightforward. If F is correct and G is correct, then $G \circ F$ is identity on the domain, and X' = X for any X in the domain. Contrapositively, if $X' \neq X$ for some X, then at least one of F or G is incorrect on that input. The round-trip test does not identify which transformation is incorrect — both must be examined when a failure occurs — but it reliably detects that something is wrong somewhere.

3.3 What Round-Trip Validates

Round-trip testing validates information preservation across the transformation pair. Every aspect of X that survives round to X' is preserved by the pair; every aspect that gets corrupted or lost is detected as a failure. This is a strong correctness condition for transformations whose purpose is to preserve information — parsers and serializers, encoders and decoders, compilers and decompilers, importers and exporters.

The validation is particularly powerful when the round-trip test is run against a diverse input corpus. A single round-trip success on one input proves only that the pair preserves that specific input; a thousand round-trip successes on a thousand structurally diverse inputs is strong evidence that the pair preserves information in general. The diversity of the input corpus directly determines the strength of the validation.

3.4 What Round-Trip Does Not Validate

Round-trip testing has three classes of failure mode it cannot detect.

First, compensating errors. If F introduces an error and G introduces a compensating error that exactly undoes F's, the round-trip succeeds while both transformations are individually wrong. This is the chief theoretical limitation. In practice it is mitigated by input diversity (compensating errors that survive one input often fail another) and by independent implementation of F and G (errors that compensate by coincidence are rare when the implementations are derived independently). The risk is not eliminated but is acceptably small in most practical settings.

Second, intentional information discarding. If F is meant to discard information that G then reconstructs from a default or canonical form, the round-trip will succeed even though information is being lost. The round-trip is checking what survives, and what was meant to be discarded is not part of what's expected to survive. This is not a flaw of round-trip testing — it correctly reflects the transformation pair's intent — but it does mean round-trip testing alone cannot validate that the discarding behavior is correct.

Third, side effects and state. Round-trip testing checks that X' = X; it does not check what F and G did along the way. If F has a side effect that should not happen (writing a log line, modifying global state, opening a file handle), round-trip testing will not detect this. If G depends on state that round-trip testing doesn't control, the test may pass or fail nondeterministically. Round-trip testing is best suited to pure or near-pure transformations; impure transformations need additional oracles to cover what round-trip cannot.

4. Levels of Round-Trip Closure

4.1 Three Levels

Round-trip closure can be specified at three levels, distinguished by the equality relation under which X' is compared to X. Each level has trade-offs in informativeness, applicability, and difficulty of achievement.

Level	Equality relation	Informativeness	Typical applicability
Exact	X' bit-identical to X	Highest — every byte preserved	Lossless serialization, persistence layers, file-format round-trips
Canonical	X' equals X after both normalized to canonical form	Medium — semantic identity, syntactic flexibility	Source code, structured data with whitespace/ordering flexibility
Semantic	X' has same meaning as X under defined interpretation	Lowest — same meaning, possibly different form	Optimization passes, refactorings, semantic-preserving transforms

Table 4.1. The three levels of round-trip closure.

4.2 Exact (Byte-Identical) Closure

Exact closure requires X' to be bit-identical to X. Every byte that was in X is in X', in the same order and with the same values; no bytes are added, removed, or changed. This is the strictest form of round-trip closure and the most informative when it can be achieved. Every information-carrying aspect of X is provably preserved, including aspects the test author may not have explicitly considered.

Exact closure is most applicable to transformations whose purpose is information preservation in its purest form: serialization/deserialization pairs, lossless compression/decompression, persistence layer save/load. The cct_repo round-trip oracle achieves exact closure between textual .cctrepo files and SQLite-backed storage: a .cctrepo file loaded into the database and unloaded back to text is byte-identical to the original. This is the cct_repo bootstrap's strongest correctness evidence — the persistence layer is provably non-lossy on every input it has been exercised against.

Exact closure is not always achievable even for correctly implemented transformations. Floating-point numbers can have multiple textual representations of the same value; XML and JSON have multiple equally-valid whitespace conventions; many file formats have valid orderings of elements that are semantically equivalent but syntactically distinct. When exact closure is not achievable, canonical closure is the next-strongest alternative.

4.3 Canonical (Normalized-Form) Closure

Canonical closure requires X' to equal X after both have been normalized to a canonical form. The normalization function maps every member of an equivalence class to a single canonical representative; two representations are canonically equal if their canonical forms are bit-identical. This accommodates representations that have multiple equally-valid forms while still providing a strong correctness signal.

Examples of canonical equivalence: source code where the canonical form is the AST (whitespace, comments, and trivial syntactic variations are normalized away); JSON where the canonical form has sorted keys and standardized whitespace; XML where the canonical form is XML Canonical Form (c14n) with standardized attribute ordering and namespace declarations. The transformation pair is correct if it preserves the canonical form, even if it doesn't preserve every byte.

Canonical closure is the natural choice when the system under test legitimately has flexibility in its output format — when there's no single "right" representation but rather an equivalence class of acceptable representations. The trade-off is informativeness: bytes that get normalized away are not validated. If the transformation pair is supposed to preserve comments and the canonical form discards them, comment-preservation is not validated by canonical closure.

4.4 Semantic (Behavioral) Closure

Semantic closure requires X' to have the same meaning as X under some defined semantic interpretation. The form may differ — even after normalization — but the meaning is preserved. This is the most permissive level of closure and the least informative as an oracle. It validates only what the semantic interpretation captures; aspects of X and X' that are outside the interpretation's scope are not validated.

Semantic closure is the appropriate level for transformations that legitimately change form while preserving meaning: compiler optimizations (the optimized code does the same thing as the original); refactorings (the refactored code is semantically equivalent to the original); query rewriting (the rewritten query returns the same results as the original). Round-trip testing at the semantic level typically involves both representations being executed or evaluated and their outputs compared, rather than the representations themselves being compared structurally.

In CCT, semantic closure appears where it must — for example, validating that two grammars are equivalent (produce the same parse trees on the same inputs) when their textual forms differ. Where exact or canonical closure is achievable, CCT prefers it; semantic closure is the fallback when stricter levels are not available.

4.5 Choosing the Right Level

The right level of closure is the strictest one that's achievable for the transformation pair under test. Stricter levels provide more informative oracles; weaker levels accommodate more legitimate variation but validate less. The choice should be made based on what the transformation pair actually preserves and what it intentionally normalizes — not on what's convenient to implement.

A common mistake is to weaken the closure level to make a test pass when the test is failing for legitimate reasons. If exact closure fails because the transformation pair has a real bug that introduces a syntactic difference, dropping to canonical closure to make the test pass hides the bug. The choice of level should be specified up front based on the transformation's contract, and stuck to.

5. Round-Trip Testing in CCT

5.1 The Substrate as Round-Trip Provider

The CCT substrate (cppcc, the CCS infrastructure, the SCB API in three languages) provides round-trip closure as a foundational service. Every SGDL grammar processed by cppcc produces a parser whose output round-trips with a corresponding serializer; every SCB (Syntax-Controlled Binary) representation has a read/write pair that round-trips through its source text. This means CCT applications inherit round-trip testing as a built-in oracle without having to construct it from scratch.

The mechanism is grammar-driven. An SGDL grammar specifies both the structure of a representation and (implicitly through that structure) how the representation is produced and consumed. cppcc generates code that parses the textual form into an SCB structure and serializes the SCB structure back to textual form. The two halves are derived from the same grammar specification, which creates a kind of structural guarantee: any divergence between them is a divergence from the shared specification, not a divergence between independently-authored components. This is closer to formal derivation than to traditional independent implementation.

5.2 cct_repo: Text-to-Database Round-Trip

The cct_repo bootstrap's central round-trip oracle is between textual .cctrepo files and the SQLite-backed database. The forward transformation parses a .cctrepo file and populates the database; the reverse transformation reads the database and writes a .cctrepo file. The oracle requires exact (byte-identical) closure: the output .cctrepo file must be bit-identical to the input.

Achieving exact closure here required deliberate design decisions. The database schema is isomorphic to the .cctrepo grammar (the im.0 isomorphic mapping) — every entity in the grammar has a table, every attribute has a column, every relationship has a foreign key. Ordering within the database (which SQLite doesn't natively guarantee) is preserved through explicit sort columns. Whitespace and formatting in the textual form are tracked via the SGDL grammar's structural specification. The result is that the persistence layer is provably non-lossy on every input the oracle has tested, which has been thousands of inputs across the bootstrap's lifetime.

This oracle was the validation mechanism for step8.6 of the cct_repo bootstrap — the initial implementation of the cct_repo CLI. The implementation was generated through Adaptive Programming from a verbal specification; the round-trip oracle was the test that confirmed the generated code worked correctly. It passed on the first attempt. Without the oracle, validating ~750 lines of generated C++ would have required manual code review or hand-authored unit tests; with the oracle, validation was a single test execution.

5.3 cppcc: Grammar Self-Round-Trip

cppcc itself uses round-trip testing recursively. SGDL is described by an SGDL grammar (the meta-grammar); cppcc processes this meta-grammar to produce the cppcc compiler itself. The cppcc compiler then processes its own meta-grammar's source text as input; the resulting parse should round-trip back to

that source text. This is the strongest possible self-consistency check: if cppcc cannot correctly process the grammar that defines its own input language, something is fundamentally wrong with cppcc.

This bootstrapping round-trip has methodological consequences. When a substrate-fix step in the cct_repo bootstrap modified cppcc (the four substrate pitfalls fixed in step8.8 through step8.10), the meta-grammar self-round-trip was the first check that confirmed the modification didn't break cppcc's foundations. The check is fast (cppcc parsing its own meta-grammar takes milliseconds) and structurally informative (any failure means cppcc's core is broken). It is one of the substrate's most useful oracles.

5.4 SCB API: Binary-to-Text Round-Trip

Every SCB-API-supported language (C++, Python, JavaScript) provides read and write operations on SCB structures that are round-trip closed against textual SGDL source. A program can read a .bfgr (binary format grammar representation) file, walk its structure, modify it, and write it back; the round-trip from binary form to walk-tree to binary form should be exact-closed, and the round-trip from textual form through binary form back to textual form should be exact-closed under the same byte-identical condition the cct_repo oracle uses.

These SCB-level round-trips are what makes the CCS Syntax-Controlled Binary User's Guide examples testable. Every example in the guide is a small program that performs an SCB operation; the guide's correctness is partly maintained by the underlying round-trip oracle: if the example's output diverges from what the round-trip should produce, the example is wrong (or the substrate has regressed).

5.5 Round-Trip as Oracle for AP-Generated Code

The most consequential use of round-trip testing in CCT is as the oracle for code generated through Adaptive Programming. When a verbal specification describes a transformation pair — say, "parse a configuration file in this format and write it back out" — the AP-generated implementation is validated by round-trip testing against a representative corpus of configuration files. If the round-trip succeeds on all inputs, the implementation is correct for those inputs; if the input corpus is sufficiently diverse, the implementation is plausibly correct in general.

This is what enables AP to generate large amounts of code with high confidence in correctness. Over a 15-day period during the cct_repo bootstrap, more than 50,000 lines of C++ were generated through Adaptive Programming. The validation was not manual code review (which would have been infeasible at that scale) but round-trip oracle invocation across the relevant test corpora. Code that passed the oracles was trusted; code that failed was returned to AP for revision. The oracle's reliability is what makes the throughput sustainable.

The key methodological insight: when round-trip closure is the contract that generated code must satisfy, the oracle's evaluation cost determines how fast code can be safely generated. Round-trip oracles are cheap to evaluate (a single execution per input, deterministic, fast); generated code is therefore cheap to validate; therefore generation can run at the rate the AI client can produce code, not at the rate the human can review code. This decoupling of generation from review is what makes AP-at-scale tractable.

6. Applicability and Limitations

6.1 When Round-Trip Testing Is the Right Oracle

Round-trip testing is well-suited when several conditions hold. The transformation pair is intended to preserve information end-to-end — parsing/serializing, encoding/decoding, compiling/decompiling, importing/exporting. Both transformations exist and are runnable as part of the test (not just one direction). The equality relation under which closure is checked is well-defined and matches the transformation pair's intent. A diverse input corpus is available or constructible. The cost of running the round-trip on each input is acceptable for the test cycle's frequency.

These conditions hold for many common software domains: data serialization formats, persistence layers, network protocols, configuration languages, source code transformations, schema evolution operations. They do not hold for domains where information is intentionally lost (compression, summarization, machine learning inference) or where the transformation is one-directional (parsing without serializing, or analyzers that have no inverse).

6.2 Limitations and What They Mean Practically

The chief limitations of round-trip testing as an oracle were named in Chapter 3: compensating errors, intentional discarding, side effects. Each has practical consequences for how round-trip testing should be deployed.

Compensating errors are mitigated by input diversity. The probability that two independent errors cancel each other on a single input is non-negligible; the probability that they cancel on a thousand diverse inputs is negligible. The practical advice is to construct or curate input corpora that exercise the transformation pair across the structural diversity of the input space — different sizes, different feature combinations, different edge cases, different syntactic variations. Round-trip testing with an impoverished corpus provides weak evidence; round-trip testing with a rich corpus provides strong evidence.

Intentional discarding is addressed by clarifying what the transformation pair is and is not meant to preserve. If F is meant to discard comments and G is meant to default them to empty, round-trip closure should be tested at the canonical level (with comments normalized away on both sides), not at the exact level. A round-trip test that succeeds because both directions agree on discarding is correctly reflecting the transformation pair's contract; the contract itself is what needs to be examined for correctness.

Side effects are addressed by either eliminating them from the transformation pair (preferring pure transformations) or by adding additional oracles that specifically check the side effects' correctness. Round-trip testing covers the data-preservation aspect; other oracles cover the side-effect aspect; together they validate the full behavior.

6.3 When Round-Trip Testing Is Not Applicable

Round-trip testing is not applicable to transformations that are inherently lossy by design — lossy compression, summarization, anonymization, redaction. The transformations have no inverse that would restore the original information, so the round-trip pattern doesn't have a meaningful closure condition.

Other oracles (similarity metrics, semantic-preservation checks, distribution-matching tests) are appropriate for such cases.

Round-trip testing is also not applicable to transformations whose intent is to produce a different representation rather than preserve information. Inference, learning, optimization that minimizes a cost function, search algorithms — these have inputs and outputs but no information-preservation relationship between them. The output's correctness is judged against the transformation's purpose (the inference is accurate, the learned model generalizes, the optimization minimizes the cost), not against the input.

Round-trip testing can be applicable but unhelpful in cases where the transformation pair is trivially correct in the round-trip sense but interesting in some other sense. A pair of inverse functions on integers (multiply by 2, then divide by 2) round-trips trivially regardless of any bugs in either function's other behavior. The oracle confirms what it confirms; if what it confirms isn't what matters about the system, the oracle is the wrong choice.

7. Comparison to Other Validation Approaches

7.1 Round-Trip Testing vs Unit Testing

Unit tests check that specific inputs produce specific expected outputs. The test author writes the expected output; the test passes when the system's actual output matches it. Unit tests are direct — they validate exactly what the test author thought to check — and limited — they validate only what the test author wrote down.

Round-trip testing and unit testing are complementary, not competing. Unit tests are appropriate when the test author knows specific input-output pairs that the system must satisfy and wants to lock those pairs in as correctness conditions. Round-trip testing is appropriate when the system's correctness condition is structural (information preservation across a transformation pair) and per-input expected outputs would be impractical to author.

In CCT, both are used. Round-trip oracles validate the structural correctness of transformation pairs at scale; unit tests in the cppcc test suite (the gtest cases added in step8.8 through step8.10) lock in specific input-output pairs for substrate-fix coverage. Each plays the role it's suited for.

7.2 Round-Trip Testing vs Property-Based Testing

Property-based testing checks that the system under test satisfies declared properties over generated inputs. The test author writes the property; the framework generates inputs; the test passes when the property holds on all generated inputs. Property-based testing generalizes unit testing — instead of locking in specific input-output pairs, it locks in properties that hold across input ranges.

Round-trip testing is a specialization of property-based testing. The property being tested is closure: $G \circ F$ is identity on the test inputs. Round-trip testing differs from general property-based testing in that the property is fixed by the transformation pair's structure (any information-preserving pair has the closure property) rather than authored per-system. This makes round-trip testing easier to apply (no per-system property authoring) but more limited in scope (it tests only the closure property).

Property-based testing frameworks (QuickCheck, Hypothesis, fast-check) can be used to implement round-trip testing — the closure property is just one property among many that the framework can check. CCT does not use a property-based testing framework in this generic sense; it uses purpose-built round-trip oracles tied to the substrate's grammar-driven infrastructure. The two approaches converge on similar coverage but reach it from different directions.

7.3 Round-Trip Testing vs Formal Verification

Formal verification proves that a system satisfies a specification across all possible inputs through mathematical argument. The proof, if correct, provides certainty that the system is correct — not just evidence from tested inputs. Formal verification is the strongest validation approach available; it is also expensive and limited to systems whose specifications can be formalized and whose proofs can be constructed.

Round-trip testing provides empirical evidence rather than proof. A round-trip oracle that succeeds on a thousand inputs has not proven anything about the thousand-and-first input; it has provided strong inductive evidence that the transformation pair preserves information for the kinds of inputs the corpus represents. The evidence is weaker than proof but is achievable at much lower cost and for systems that resist formal specification.

In a sense, round-trip testing is the engineering practitioner's substitute for formal verification on the subset of correctness conditions (information preservation) where round-trip is applicable. Where formal verification is feasible and affordable, it is stronger. Where it is not, round-trip is the next-best option for the same correctness concern.

7.4 Round-Trip Testing vs Code Review

Code review validates that human reviewers consider the code correct upon reading it. Code review catches issues that no automated check would catch (questionable design choices, unclear naming, latent maintainability problems) and misses issues that humans systematically overlook (subtle off-by-one errors, race conditions, edge cases the reviewer didn't think to check).

Round-trip testing covers a different validation surface. Round-trip oracles catch information-loss errors with high reliability; they do not catch design problems, naming problems, or maintainability issues. The two approaches are entirely complementary and should both be used where appropriate.

The methodological implication for AP, however, is significant. Code review at the line-by-line level is the validation bottleneck for traditional development — code that hasn't been reviewed cannot be safely merged. Round-trip testing replaces this bottleneck for information-preserving transformations: code that passes the oracle can be safely merged without line-by-line review, even if it was generated rather than written. This is what enables AP throughput at the scale observed in the cct_repo bootstrap.

8. Constructing Round-Trip Tests

8.1 The Basic Construction

Constructing a round-trip test requires four elements: an input corpus of representative X values; the forward transformation F; the reverse transformation G; and a comparison procedure for X' against X under the chosen equality relation. The test is the loop that runs each X through F to produce Y, runs Y through G to produce X', and compares X' to X.

```
for each X in corpus:
    Y  = F(X)
    X' = G(Y)
    assert equal(X, X', equality_relation)
```

The construction is mechanical once the four elements are settled. The substantive work is in choosing them well — particularly the input corpus, since the test's strength is directly proportional to the corpus's diversity.

8.2 The Input Corpus

Three approaches to corpus construction are common. Curated corpus: the test author selects inputs that exercise the transformation pair's diverse cases — typical inputs, edge cases, regression cases from past bugs, inputs that stress specific structural features. Generated corpus: a generator produces inputs according to a grammar or distribution — the generator's quality determines the corpus's coverage. Field corpus: real-world inputs collected from actual usage — these have the advantage of representing what the transformation pair will see in production but are not always available.

In CCT specifically, generated corpora are natural because the inputs are themselves grammar-conformant — an SGDL grammar can be used to generate diverse grammar-conformant inputs for round-trip testing of whatever the grammar describes. This is a virtuous cycle: the grammar enables both the transformation pair (parsing and serializing) and the corpus generator (producing grammar-conformant inputs). The test infrastructure scales because the grammar does the work.

8.3 The Comparison Procedure

The comparison procedure depends on the closure level. For exact closure, byte-by-byte comparison; modern languages and tools provide this trivially. For canonical closure, both X and X' are run through the normalization function and the normalized forms are compared byte-by-byte. For semantic closure, both are interpreted under the semantic interpretation and the interpretations' results are compared — this is more involved and may require the semantic interpretation itself to be a verified component.

The comparison's failure mode matters. When X' ≠ X, the test should fail loudly with a diff that helps diagnose where the divergence occurred. Naive byte-comparison failure messages are unhelpful for large inputs; smarter diff tools (text diff, JSON diff, AST diff) make round-trip test failures actionable.

8.4 Test Cycle Integration

Round-trip tests should be integrated into the development cycle's automated test suite, run on every change, and treated as blocking failures. The economic argument: each round-trip test invocation is cheap, and a failing oracle on a small input is much easier to diagnose than the same failure surfacing later through a downstream consumer. The methodological argument: in CCT specifically, round-trip oracles are the validation mechanism for AP-generated code; the development cycle's correctness guarantee depends on them running.

Test cycle integration also enables continuous corpus growth. When a bug is found that the round-trip test missed, the inputs that triggered the bug are added to the corpus, and the corpus is permanently larger and more diverse. Each bug found becomes future bug prevention. Over time, the corpus accumulates a record of the transformation pair's history and serves as living regression protection.

8.5 What to Do When the Round-Trip Fails

Round-trip test failures are diagnostic rather than self-explanatory: the test tells you that something is wrong but not where. The diagnostic procedure is to narrow down the failure to the smallest input that reproduces it (delta-debugging or manual bisection), then examine the intermediate value Y to determine whether F or G or both produced the divergence.

If Y looks correct but X' is wrong, G is the suspect. If Y looks wrong, F is the suspect. If both look wrong in compensating ways, the compensating-errors case has surfaced and both transformations need examination. In practice, the smallest-failing-input technique is the workhorse: the smaller the input, the easier the manual examination of F's output and G's input.

Once the failing transformation is identified and fixed, the failing input is added to the corpus permanently. The round-trip oracle becomes stricter after every fix, since the corpus the oracle exercises has grown by an input the system previously failed.

8.6 Closing Note

Round-trip testing is one of the most cost-effective oracles available for the subset of correctness conditions to which it applies. It is cheap to construct (the transformation pair already exists), cheap to run (deterministic, fast), and produces strong evidence per invocation. Where it applies, it should be used. Where it doesn't apply, other oracles take its place — but the niches where it does apply are large and growing as more systems are built around information-preserving transformation pairs, including the AP-generated systems that CCT produces.

In CCT specifically, round-trip testing is what makes the methodology's scale achievable. It is the oracle that turns AP-generated code from "plausible but unverified" into "validated against structural

correctness conditions." Without it, the throughput numbers observed in the cct_repo bootstrap would not be defensible. With it, those numbers are not only defensible but reproducible.