Error Messages as Documentation

Designing compiler diagnostics that teach, not just report

Lateralus Language

bad-antics · September 2024 · Lateralus Language Research

ABSTRACT Compiler error messages are the most-read documentation a language produces. A programmer who encounters an error reads the message before consulting any other source; if the message is unclear, the debugging session begins poorly. Lateralus treats error messages as first-class documentation artifacts subject to the same review standards as the language specification. This paper describes the four principles of the Lateralus error message design system, the tooling that enforces them, and a user study comparing message quality against Rust, TypeScript, and Python.

1. Why Error Messages Are Documentation

API documentation explains how to use a feature correctly. Error messages explain what went wrong and how to fix it — documentation for the failure case. Yet most language teams treat error messages as an afterthought: a string appended to a syntax rule or type check failure, written once and never reviewed.

The cost of poor error messages is high. A user study by Becker et al. (2019) found that novice programmers spend 40-60% of their debugging time interpreting compiler output. A message that misidentifies the root cause or fails to suggest a fix doubles or triples the time to resolution.

Lateralus allocates dedicated engineering time to error message quality. Each error code has a message template, an extended explanation, one or more code examples of the wrong pattern, and one or more code examples of the correct pattern. The message template is separate from the code that detects the error; changing one does not require changing the other.

2. The Four Principles

2.1 Identify the Root Cause, Not the Symptom

A type mismatch error typically has a root cause far from the reported location: a function was defined with the wrong return type, a variable was assigned the wrong value, or an import resolved to the wrong module. The symptom is the mismatch at the call site; the root cause is the definition site.

Lateralus's type checker propagates origin metadata: when a type is inferred from an expression, the expression's source location is attached to the type. When a mismatch occurs, the error message shows both the location of the conflict and the location where the conflicting type was established.

```
error[E0012]: type mismatch
  --> src/handler.lt:42:5
   |
42 |     |?> serialize_json    // expects Json, got XmlDoc
   |     ^^^^^^^^^^^^^^^^^ pipeline stage expects Json
   |
note: XmlDoc introduced here
  --> src/handler.lt:38:5
38 |     |>  parse_xml         // returns XmlDoc
   |     ^^^^^^^^^^ this returns XmlDoc, not Json
hint: use convert::xml_to_json to bridge the types
```

2.2 Suggest a Fix

Every error message that has a known common fix includes a 'hint' line with the fix. When the fix can be applied automatically, the hint line ends with (auto-fixable) and the compiler's --fix flag applies it without user intervention.

2.3 Show, Don't Tell

Abstract descriptions of type rules are less useful than concrete examples. Error messages include a 'see also' block with a minimal code example that demonstrates the correct pattern, taken from the error's documentation entry in the error catalog.

2.4 One Error at a Time for Beginners

For users in beginner mode (--beginner flag), the compiler emits only the first error and its full explanation before stopping. Expert mode (default) emits all errors grouped by file.

3. Error Code Catalog

Every Lateralus error has a code in the format ENNNN. The catalog is a searchable document (available at lateralus.dev/errors/) with one page per code containing: the error class, a natural-language explanation, the bad pattern, the good pattern, and the rationale for the rule that triggered the error.

```
E0001-E0099:  Syntax errors
E0100-E0199:  Type errors
E0200-E0299:  Pipeline operator errors
E0300-E0399:  Lifetime / borrow errors
E0400-E0499:  Name resolution errors
E0500-E0599:  FFI / unsafe errors
E0600-E0699:  Module system errors
E0700-E0799:  Async / concurrency errors
```

Each error code has a designated owner (a compiler team member) responsible for keeping the message and documentation current. The owner is listed in the catalog and cc'd on any issue report related to that error.

4. Pipeline-Specific Errors

Pipeline errors (E0200-E0299) deserve special attention because they are the errors most specific to Lateralus and most likely to confuse programmers migrating from other languages.

4.1 E0201: Operator Variant Mismatch

Emitted when a stage in a |?> chain returns a total value (T) instead of a Result<T, E>:

```
error[E0201]: |?> requires a Result-returning stage
  --> src/main.lt:10:5
   |
10 |     |?> enrich
   |     ^^^^^^^^^^ this stage returns User, not Result<User, _>
hint: if enrich cannot fail, use |> instead of |?>
```

4.2 E0202: Error Type Divergence

Emitted when consecutive |?> stages return incompatible error types:

```
error[E0202]: error types diverge in pipeline
  --> src/main.lt:14:5
   |
12 |     |?> parse      // returns Result<Parsed, ParseError>
   |     ------------- error type here: ParseError
14 |     |?> validate   // returns Result<Valid, ValidationError>
   |     ^^^^^^^^^^^^^ error type here: ValidationError
```

```
hint: use |?> with an error converter: |?> validate.map_err(From::from)
```

5. The Lint Layer

Above the error layer (which reports definite mistakes) is a lint layer (which reports likely mistakes and style violations). Lints follow the same message structure as errors but are emitted as warnings by default. All lints are suppressible with a #[allow(lint_name)] attribute.

Pipeline-specific lints include:

- redundant_ok_wrap: a stage in a |?> chain always returns Ok(x) and could be replaced by a total |> stage.
- shadowed_error: a |~> recovery stage discards the original error without logging it.
- long_pipeline: a pipeline has more than 12 stages; consider splitting it into named sub-pipelines.
- blocking_in_async: a total stage in an async |>> pipeline calls a blocking function without async::spawn_blocking.

6. User Study Results

We ran a controlled user study with 40 participants (20 experienced, 20 novice Lateralus users) and presented 10 error scenarios each. Participants used either the Lateralus error output or an equivalent error from Rust, TypeScript, or Python, randomly assigned. We measured time to identify the fix and number of documentation lookups required.

Metric	Lateralus	Rust	TypeScript	Python
Mean time to fix (expert)	42 s	67 s	89 s	120 s
Mean time to fix (novice)	95 s	180 s	210 s	310 s
Documentation lookups/error	0.3	1.1	1.8	2.4
Fix identified correctly	97%	81%	74%	62%

The Lateralus numbers are best-in-class for all metrics. The largest gains are for novice users, where the 'show, don't tell' principle and the beginner mode have the most impact.

7. Tooling: Error Message Regression Tests

Error messages are covered by a dedicated test suite that asserts the exact text of every error the compiler emits for a given erroneous input. Any change to an error message that does not update the corresponding test causes a test failure.

```
// Regression test for E0201
#[test]
fn test_e0201_variant_mismatch() {
    let source = r#"
        let result = x |?> enrich
        // enrich returns User, not Result<User, _>
    "#;
    let errors = compile_and_collect_errors(source);
    assert_eq!(errors[0].code, "E0201");
    assert!(errors[0].message.contains("|?> requires a Result"));
    assert!(errors[0].hints[0].contains("use |> instead"));
}
```

The test suite has 1,240 test cases, one for each error code plus additional tests for common variant messages. A release is blocked if the error message test suite does not pass at 100%.

8. Conclusion

Error messages are documentation for the failure case. Treating them as first-class artifacts — with ownership, review standards, regression tests, and a user study — produces measurably better debugging experiences, especially for novice users.

The investment pays off multiplicatively: a programmer who resolves their first error quickly is more likely to continue learning the language. The quality of the first error message a new user sees is a significant factor in language adoption.