Lateralus: A Pipeline-Native Language

Overview of a systems language built around typed data pipelines as the primary abstraction

Lateralus Language

bad-antics · April 2026 · Lateralus Language Research

ABSTRACT Lateralus is a statically typed systems programming language that makes data pipelines a first-class language construct rather than a library abstraction. This overview introduces Lateralus to readers unfamiliar with the language: its motivation, its core constructs, its intended application domains, and how it compares to existing systems languages. No prior knowledge of Lateralus is assumed.

1. Motivation

Most programs are fundamentally pipelines: data flows in, is transformed through a sequence of stages, and flows out. Yet most programming languages treat pipelines as a pattern to be implemented by the programmer — a chain of function calls, a sequence of method invocations, or a Bash pipe.

Lateralus treats the pipeline as the primary syntactic and semantic unit. The pipeline operator |> is not syntactic sugar for function application; it is a distinct evaluation form with its own typing rules, optimization opportunities, and tooling support. This distinction matters: a language that understands pipelines can reason about them in ways a language with method chaining cannot.

2. Hello, Lateralus

A minimal Lateralus program that reads a file, filters its lines, and prints the matches:

```
fn main() -> Result<(), IoError> {
    std::args().skip(1).first()
        |?> fs::read_to_string
        |>  str::lines
        |>  filter(|line| line.contains("error"))
        |>  iter::for_each(println)
}
```

Each stage after the first receives the output of the previous stage. The |?> operator propagates errors without explicit match or ? syntax — it is the pipeline equivalent of Rust's ? operator.

3. The Four Pipeline Operators

Lateralus has four pipeline operators, each with distinct semantics:

```
Operator  Name        Semantics
|>        Total       Always succeeds; A → B
|?>       Fallible    May fail; Result<A,E> → Result<B,E>
|>>       Async       Concurrent map; Stream<A> → Stream<B>
|>|       Collect     Terminates a stream; Stream<A> → Vec<A>
```

The choice of operator is explicit and meaningful: a programmer reading |?> knows the stage may fail and the error is propagated. A programmer reading |>> knows the operation is concurrent. The operator is documentation.

4. Types and Inference

Lateralus uses Hindley-Milner type inference. Most programs require no type annotations; the compiler infers all types from usage. Annotations are required only at module boundaries (public functions) and when inference is ambiguous:

```
-- Inferred: no annotations needed
```

```
fn double_all(xs: Vec<i32>) -> Vec<i32> {
    xs |> map(|x| x * 2)
}

-- Annotation required: return type is a type alias
pub fn parse_config(s: &str) -> Result<Config, ConfigError> {
    s |> toml::parse |?> Config::from_toml
}
```

5. Ownership Without Pain

Lateralus uses ownership-based memory management: no garbage collector, no manual free. The ownership model is similar to Rust's but with several ergonomic improvements:

- **Move by default in pipelines**: pipeline stages automatically move their input, eliminating the need for explicit .clone() in most cases.
- **Implicit borrows for read-only stages**: stages that only read their input receive an implicit immutable borrow rather than consuming the value.
- **Borrow inference**: the compiler infers borrow kinds in most cases, requiring explicit & and &mut; only when the borrow kind is ambiguous.

6. Application Domains

Lateralus is designed for three primary domains:

```
Domain                  Key features used
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
Systems / OS kernel     no_std, ownership, inline asm
Security tooling        typed schemas, scope types, pipelines
Data processing         async pipelines, streaming, WASM target

Secondary domains:
  Network services      async, TLS, typed protocols
  Embedded systems      no_std, RISC-V target, deterministic timing
  Language tooling      LSP integration, playground, formatter
```

The language does not target mobile, GUI, or game development — these domains have existing ecosystems (Swift/Kotlin, React/Qt, Unity) that Lateralus does not aim to displace.

7. Comparison with Similar Languages

How Lateralus relates to languages it is frequently compared to:

```
Language   Comparison
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
Rust       Same memory model; Lateralus adds pipeline syntax
           and effect types. Rust has larger ecosystem.
Elixir     Both pipeline-oriented; Elixir is GC'd and dynamic.
           Lateralus is systems-level and statically typed.
Haskell    Both have strong type systems; Lateralus is eagerly
           evaluated, imperative-first, and explicitly effectful.
C          C has no safety or pipelines. Lateralus is C's successor
           for applications that need both safety and performance.
```

8. Current Status and Roadmap

Lateralus is at version 1.4 (stable). The current status:

```
v1.4 (current, stable):
  ✓ RISC-V and x86-64 backends
  ✓ Core type system and inference
  ✓ Pipeline operators (|>, |?>, |>>, |>|)
  ✓ Ownership and borrows
  ✓ Package registry and ltlup toolchain manager
  ✓ LSP (ltl-lsp) with pipeline inlay hints

v2.0 (planned, 2027):
  → AArch64 backend
  → Effect type system (stable)
  → Formal verification framework
  → WebAssembly component model support
```