Toward a Pipeline-Native Language: Design Rationale

Why existing pipeline proposals fall short and what first-class pipelines look like

Lateralus Language

bad-antics · September 2023 · Lateralus Language Research

ABSTRACT We survey existing pipeline operator implementations across F#, Elixir, OCaml, and the TC39 proposal for JavaScript. We identify common limitations — syntactic sugar over function application, no error propagation, no async integration — and propose Lateralus, a language where the pipeline operator is a first-class semantic construct with variants for error handling, async streaming, and parallel fan-out. We demonstrate through empirical measurement and formal argument that each variant addresses a distinct failure mode in existing designs, and that treating pipelines as a first-class form rather than library sugar enables compiler optimizations not achievable otherwise.

1. Motivation

Every modern program moves data through stages: parse, validate, transform, serialize, persist. The pipeline metaphor is universal. Yet the programming languages most commonly used for such tasks — Python, JavaScript, Java — express pipelines awkwardly, as deeply nested function calls that must be read inside-out, or as long sequences of intermediate variable assignments that clutter scope. The cognitive cost is real: studies of program comprehension consistently show that nested call depth beyond three levels substantially increases the time required for correct understanding.

Functional languages recognized this decades ago. ML-family languages pioneered the pipeline operator |> as reverse function application, enabling linear reading of data-transformation chains. Elixir adopted and popularized this idea on the BEAM runtime. OCaml standardized its own variant. The TC39 committee has debated a JavaScript pipeline operator for years. Yet all these designs share a critical property: the pipeline operator is syntactic sugar over ordinary function application. It is desugared during parsing and disappears entirely from the language's intermediate representation. This makes it impossible for the compiler to reason about pipelines as units, apply pipeline-specific optimizations, or provide pipeline-aware error messages.

Lateralus takes a different position. The pipeline operator is a first-class syntactic form that survives into the intermediate representation. The compiler pipeline (no pun intended) is organized around pipeline stages as the unit of optimization. Error messages are phrased in terms of which stage produced an incompatible type. Tracing and profiling infrastructure annotates output by stage. This paper documents the design decisions behind that choice, beginning with a survey of existing approaches and their limitations.

The rest of this paper is organized as follows. Sections 2 through 5 survey F#, Elixir, OCaml, and the TC39 JavaScript proposal respectively, analyzing each on a common set of criteria. Section 6 synthesizes the common failure patterns we observe. Sections 7 through 10 describe the Lateralus design: what first-class means, the four operator variants, grammar decisions, and type-system integration. Sections 11 through 15 cover error propagation, async integration, parallel fan-out, toolchain implications, and performance. Sections 16 and 17 cover related work and conclusions.

Throughout we use a uniform benchmark suite of five pipeline-heavy programs: a JSON processing chain, an HTTP request pipeline, a numeric ETL (extract-transform-load) pipeline, a recursive-descent parser expressed as a pipeline, and a stream-processing aggregation. These programs are implemented in each surveyed language (or its idiomatic equivalent) so that claims about expressiveness and performance are grounded in concrete measurements rather than intuition.

2. Survey: F# Pipelines

F# introduced the pipeline operator |> as part of its initial design, drawing on earlier ML tradition. In F#, x |> f is defined as f x and the operator is implemented as a simple two-argument infix function in the core library: let (|>) x f = f x. There is no compiler-internal representation of the pipeline form; by the time the compiler's front-end hands off to the elaborator, pipelines have already been desugared into ordinary function application.

```
// F# pipeline: desugared immediately to nested application
let result =
    input
    |> List.filter isValid
    |> List.map transform
    |> List.sortBy key
    |> List.take 10

// After elaboration, the compiler sees:
let result = List.take 10 (List.sortBy key (List.map transform (List.filter isValid input)))
```

The F# approach works well for simple linear data transformations and is deeply integrated into the idiom of the language. The F# standard library is designed with pipeline use in mind: every collection function takes its subject as the last argument, enabling point-free pipeline composition. This convention, sometimes called the 'data-last' convention, is essential to making |> ergonomic. Without it, the programmer must write partial application at every stage.

However, F# pipelines have no error-propagation semantics. The operator is blind to Result<'a, 'e> and Option<'a> types. A programmer who wants to propagate errors through a pipeline must use the computation expression syntax (F#'s monad comprehension mechanism) or write explicit pattern-match unwrapping at every stage. Neither approach preserves the linear, left-to-right reading that motivates pipelines in the first place. The computation expression syntax, while powerful, introduces its own cognitive overhead and requires understanding of monadic bind.

```
// F#: error propagation requires computation expression, breaking pipeline style
let processRequest (req : HttpRequest) : Result<Response, AppError> =
    result {
        let! parsed   = parseRequest req          // explicit bind
        let! validated = validateRequest parsed
        let! response  = executeRequest validated
        return formatResponse response
    }

// Desired but not available in F#:
// req |?> parseRequest |?> validateRequest |?> executeRequest |> formatResponse
```

F# also lacks any native async integration with the pipeline operator. Asynchronous operations require wrapping in async { ... } computation expressions, completely abandoning pipeline syntax. The task { ... } CE in F# 6 improved matters somewhat but does not enable a pipeline style for async chains. Additionally, F# provides no parallel fan-out primitive within the pipeline idiom; parallel execution requires explicit use of Async.Parallel or Task.WhenAll, again abandoning the pipeline form.

3. Survey: Elixir Pipelines

Elixir's pipeline operator |> is syntactically identical to F#'s but operates in a dynamically typed context on the BEAM runtime. Elixir's operator inserts the left-hand expression as the first argument of the right-hand function call, rather than as the sole argument. This is the 'data-first' convention, the mirror image of F#'s data-last. Elixir's standard library and all community libraries are written with data-first in mind, making pipeline composition natural throughout the ecosystem.

```
# Elixir pipeline: data inserted as first argument of each stage
result =
  input
  |> Enum.filter(&valid?/1)
  |> Enum.map(&transform/1)
  |> Enum.sort_by(&key/1)
  |> Enum.take(10)

# With arity > 1, extra args follow the implicit first arg:
"hello world"
|> String.split(" ")       # => String.split("hello world", " ")
|> Enum.map(&String.upcase/1)
```

Elixir's dynamic typing eliminates an entire category of pipeline error: there are no type-mismatch failures at compile time, only at runtime. This is a double-edged sword. Pipelines never fail to compile due to type errors; they can fail at runtime with cryptic FunctionClauseError exceptions that do not clearly indicate which pipeline stage produced the incompatible value. Dialyzer, Elixir's gradual type analysis tool, can sometimes catch such errors but its coverage of pipeline expressions is incomplete and its error messages are notoriously difficult to interpret.

Elixir does have a convention for error propagation in pipelines: the with macro, which provides pattern-matching-based short-circuit semantics. However, with is syntactically separate from the pipeline operator and cannot be used inside a pipeline chain. The two idioms do not compose. In practice, Elixir programmers either abandon pipeline style entirely when error handling is needed, or write wrapper functions that convert error values back into plain values (and risk swallowing errors silently).

```
# Elixir: with macro for error propagation — NOT composable with |>
def process(input) do
  with {:ok, parsed}    <- parse(input),
       {:ok, validated} <- validate(parsed),
       {:ok, result}    <- execute(validated) do
    {:ok, format(result)}
  else
    {:error, reason} -> {:error, reason}
  end
end

# Cannot write: input |> parse() |> validate() |> execute() |> format()
# without losing error propagation.
```

The BEAM runtime gives Elixir genuine concurrency strengths, but these are not exposed through the pipeline operator. Running multiple pipeline stages concurrently requires explicit Task.async and Task.await calls, or use of the GenStage/Flow libraries, which have entirely different compositional idioms. The pipeline operator in Elixir remains a purely sequential, purely synchronous form despite running on a runtime purpose-built for concurrency.

4. Survey: OCaml Pipelines

OCaml standardized the pipeline operator |> in version 4.01 (2013), defined as let (|>) x f = f x in the Stdlib module. As in F#, it is a library function desugared before any optimization. OCaml's type system is stronger than Elixir's and comparable to F#'s: the operator is polymorphic of type 'a -> ('a -> 'b) -> 'b, and type errors at pipeline stages are caught at compile time through Hindley-Milner inference.

```
(* OCaml pipeline: library function, strongly typed *)
let result =
  input
  |> List.filter is_valid
  |> List.map transform
  |> List.sort_uniq compare
  |> List.filteri (fun i _ -> i < 10)

(* Type error caught at compile time: *)
let bad = 42 |> List.map transform
(* Error: This expression has type int but an expression of type 'a list was expected *)
```

OCaml's error messages for pipeline type errors are significantly better than Elixir's runtime crashes, but they are still expressed in terms of the desugared form. The compiler does not know the user intended to write a pipeline; it reports errors as if the user had written nested application. A type mismatch in the fourth stage of a ten-stage pipeline produces an error pointing to the desugared application site, which often corresponds to a confusing position in the source code after OCaml's reformatting.

OCaml also lacks error-propagation and async pipeline variants. The result type (introduced in 4.03) and the let* binding operator (4.08) together enable monadic error handling, but not through the |> operator. The Effect system (5.x) introduces algebraic effects and multi-shot continuations that can implement async in principle, but integration with the pipeline operator is absent from the standard library and from the main body of community libraries. Each library (Eio, Lwt, Async) has its own combinator style that does not compose with |>.

```
(* OCaml: let* for error propagation, separate from |> *)
let process input =
  let* parsed    = parse input in
  let* validated = validate parsed in
  let* result    = execute validated in
  Ok (format result)

(* The two idioms cannot be mixed:
   input |> parse |> validate |> execute  -- loses error propagation
   let* style                              -- loses pipeline readability *)
```

One OCaml-specific concern is the interaction between the pipeline operator and OCaml's labeled and optional arguments. A function with labeled arguments cannot be used at a pipeline stage without explicit partial application, because the pipeline operator passes the left-hand value positionally. This is a sharp edge in practice: much of the OCaml standard library uses labeled arguments for clarity, but pipeline users must write adapter functions or use explicit Fun.flip calls to work around the convention mismatch. Lateralus eliminates this problem by making the pipeline operator aware of argument position at the type level.

5. Survey: TC39 JavaScript Pipeline Proposal

The TC39 pipeline operator proposal for JavaScript has a long and contentious history. The proposal has existed in some form since 2015 and has gone through multiple competing designs, committee debates, and stage regressions. As of this writing (September 2023) the 'Hack-style' pipeline is the surviving proposal, having displaced the earlier 'F#-style' and 'Smart-mix' variants. In Hack-style pipelines, the topic variable % (or ^^ depending on the draft) receives the left-hand value and must appear explicitly in the right-hand expression.

```
// TC39 Hack-style pipeline (proposal, not yet standardized)
const result = input
  |> %.filter(isValid)
  |> %.map(transform)
  |> %.sort((a, b) => key(a) - key(b))
  |> %.slice(0, 10);

// vs F#-style (rejected alternative):
// const result = input |> filter(isValid) |> map(transform) |> ...
```

The Hack-style design allows arbitrary expressions on the right-hand side, not just function references. This gives the operator more expressive power than F#-style (which is limited to unary function references) but introduces the topic variable as a new concept that programmers must learn. The proposal has been criticized for adding syntactic weight without the clarity benefits that motivated pipeline operators in the first place. The syntax % is also used by the remainder operator, creating potential confusion.

JavaScript's dynamic typing means the TC39 proposal carries no type-safety guarantees. Like Elixir, pipeline errors manifest as runtime exceptions. Unlike Elixir, TypeScript type-checking does not meaningfully handle the topic variable: the TypeScript team has not committed to supporting the proposal until it reaches TC39 Stage 3, and the type inference challenges posed by the topic variable are non-trivial. At the time of writing, TypeScript 5.2 does not support the pipeline operator at all.

```
// TC39 proposal: no error propagation semantics
// A throwing stage aborts the entire pipeline with an uncaught exception
const result = fetchUser(id)
  |> validateUser(%)     // throws if invalid
  |> enrichUser(%)
  |> formatUser(%);

// Must use try/catch externally, cannot express error routing per-stage
try {
  const result = fetchUser(id) |> validateUser(%) |> ...;
} catch (e) {
  // Which stage failed? The stack trace may not tell you.
}
```

The TC39 proposal has no async or concurrent semantics. The committee has explicitly deferred async pipeline integration to a future proposal, acknowledging that the interaction between |> and await is complex. In practice this means the pipeline operator cannot be used with async/await chains without manually inserting await at each stage, breaking the linear-reading benefit. The committee's position is that this is a limitation worth accepting to ship a simpler proposal; Lateralus's position is that async integration is a first-order requirement, not a future nicety.

6. Common Failure Patterns Across Existing Designs

Having surveyed four existing pipeline designs, we can identify three failure patterns that appear consistently across all of them. These are not coincidental; they follow directly from the design decision to treat |> as syntactic sugar.

6.1 Failure Pattern: Error Propagation Abandonment

Every surveyed language forces programmers to choose between pipeline style and error propagation. F# uses computation expressions; Elixir uses with; OCaml uses let*; JavaScript uses try/catch. In every case, the error-handling idiom is syntactically incompatible with the pipeline idiom. Programmers who want both must either nest one inside the other (breaking linear readability) or abandon one entirely (losing either safety or clarity). This is not a library design problem; it is a language design problem. The pipeline operator needs a variant that is inherently error-aware.

6.2 Failure Pattern: Async Abandonment

Every surveyed language forces programmers to choose between pipeline style and async execution. Async in F# requires computation expressions; in Elixir requires task spawning; in OCaml requires Lwt/Eio combinators; in JavaScript requires explicit await insertion. None of these integrate with the |> operator. The result is that real-world pipelines that involve I/O — which is most real-world pipelines — cannot use the pipeline idiom consistently. Lateralus's |>> operator addresses this directly.

6.3 Failure Pattern: Parallelism Abandonment

None of the surveyed languages provide a pipeline operator variant for parallel fan-out. When a pipeline stage can be applied independently to multiple inputs, or when multiple independent stages can run simultaneously, the programmer must leave the pipeline idiom entirely and use language-specific concurrency APIs. This forces a context switch in the programmer's mental model at exactly the point where data-flow thinking is most natural. Lateralus's |>| operator provides parallel fan-out within the pipeline idiom.

6.4 Failure Pattern: Opaque Compiler Treatment

Because all surveyed languages desugar |> before optimization, the compiler cannot perform pipeline-aware optimizations. Stage fusion (combining adjacent stages into a single traversal to eliminate intermediate allocations) must be discovered by general-purpose optimizer passes that have no knowledge of the user's intent. Dead-stage elimination (removing stages whose output is unused) requires whole-program analysis. Pipeline-specific error messages are impossible because the compiler has already discarded the pipeline structure. Lateralus retains pipeline structure through to code generation, enabling all of these.

7. What First-Class Pipelines Mean

We use the term 'first-class pipeline' to mean a pipeline operator that satisfies three properties: (1) the pipeline form is distinct in the language's intermediate representation from ordinary function application; (2) the compiler performs passes that specifically recognize and transform pipeline forms; and (3) the runtime (or emitted code) may execute pipeline stages in ways that differ from ordinary function application, such as fusing allocations or parallelizing execution.

Property (1) is the foundational requirement. If the pipeline operator is desugared to function application before the IR, properties (2) and (3) become very difficult to achieve reliably. A general-purpose optimizer might recover some pipeline structure through pattern-matching on the desugared form, but this is fragile and incomplete. Lateralus's IR has a Pipeline node distinct from Apply; compiler passes

can inspect and transform it directly.

```
// Lateralus IR (simplified textual notation)
// Source:  input |> validate |> transform |> serialize

Pipeline [
  Stage { fn: validate,   input: #0 },
  Stage { fn: transform,  input: prev },
  Stage { fn: serialize,  input: prev },
]

// vs. what a desugaring language would produce:
Apply {
  fn: serialize,
  arg: Apply { fn: transform, arg: Apply { fn: validate, arg: #0 } }
}
```

Property (2) enables the suite of pipeline-specific optimizations described later in this paper. Stage fusion, dead-stage elimination, and error-short-circuit hoisting all operate on the Pipeline IR node directly. They are implemented as dedicated compiler passes that run after type-checking and before code generation. Because these passes see the user's intent (a pipeline of stages) rather than its desugaring (nested application), they can make decisions that general-purpose passes cannot.

Property (3) enables the |>| parallel fan-out variant and the |>> async streaming variant. Both require runtime behavior that fundamentally differs from sequential function application: the parallel variant spawns tasks and joins them; the async variant drives a coroutine scheduler. These behaviors cannot be expressed as simple syntactic desugaring without introducing complex runtime machinery that the programmer cannot see. Lateralus makes the machinery explicit in the type of the operator itself.

It is worth emphasizing what first-class does not mean. It does not mean that pipelines are a new data type at the value level — a Lateralus pipeline expression evaluates to the type of its final stage, not to a 'pipeline object'. It does not mean that pipelines are inherently lazy or that stages are reified as closures at runtime. First-class is a property of the compiler's treatment, not of the runtime representation. This distinction matters because it means Lateralus pipelines have zero overhead compared to hand-written equivalent code: the IR nodes are eliminated during code generation, leaving exactly the code a careful programmer would have written by hand.

8. The Four-Variant Pipeline Design

Lateralus provides four pipeline operator variants, each addressing a distinct use case. The variants are: |> (basic), |?> (error-propagating), |>> (async streaming), and |>| (parallel fan-out). Together they cover the space of common pipeline patterns identified in the failure analysis above. Each variant is a distinct syntactic form with distinct typing rules and distinct code generation strategies.

8.1 Basic Pipeline: |>

The basic pipeline operator has the same semantics as F# and OCaml: x |> f evaluates to f x. The difference is that the IR retains the pipeline form. The typing rule is standard: if x : T and f : T -> U, then x |> f : U. All four operators use Hindley-Milner inference, so stage types are inferred from context and never need to be written explicitly.

```
// Basic pipeline: sequential, no error propagation
let result =
    read_file("data.csv")
    |> parse_csv
    |> filter(fn row => row.age > 18)
    |> map(fn row => row.name)
```

```
    |> sort
    |> take(100)
```

8.2 Error Pipeline: |?>

The error-propagating pipeline requires that each stage returns Result<T, E>. If a stage returns Err(e), the error is propagated to the end of the pipeline without executing any subsequent stages. The typing rule requires all stages to share the same error type E. This is the Kleisli-composition semantics proven in our companion paper on pipeline calculus.

```
// Error pipeline: short-circuits on first Err
let result : Result<Response, AppError> =
    request
    |?> parse_request        // : Request -> Result<Parsed, AppError>
    |?> validate_auth        // : Parsed -> Result<Authed, AppError>
    |?> fetch_data           // : Authed -> Result<Data, AppError>
    |?> format_response      // : Data -> Result<Response, AppError>

// Type error: mismatched error types caught at compile time
// let bad = x |?> f |?> g  where f : A -> Result<B, Err1>
//                          and   g : B -> Result<C, Err2>  -- ERROR
```

8.3 Async Pipeline: |>>

The async pipeline requires that each stage is an async function returning a future. The pipeline drives a coroutine scheduler: each stage is awaited in turn, and the pipeline expression itself evaluates to an async value that must be awaited by the caller. The scheduler is pluggable; the standard library provides a work-stealing executor, but custom executors (e.g., for embedded systems) can be registered.

```
// Async pipeline: each stage is awaited
let response : Async<Result<Page, Error>> =
    url
    |>> fetch_url             // : Url -> Async<Body>
    |>> parse_html            // : Body -> Async<Dom>
    |>> extract_links         // : Dom -> Async<List<Url>>

// Async + error pipeline can be combined:
let result =
    url
    |>> fetch_url             // Async stage
    |>> parse_html
    |?> validate_structure    // Error stage (synchronous, wrapped)
    |>> store_result
```

8.4 Parallel Fan-Out: |>|

The parallel fan-out operator takes a value and a list of functions and applies each function to the value concurrently, collecting results into a list. It is syntactically written with a list literal on the right-hand side. The typing rule requires all functions in the list to accept the same input type; their return types may differ (the result is a heterogeneous tuple when the types differ, and a homogeneous list when they are all the same).

```
// Parallel fan-out: apply multiple functions concurrently
let [count, total, avg] =
    dataset
    |>| [count_records, sum_values, compute_average]

// Can compose with subsequent stages:
```

```
let report =
    dataset
    |>| [count_records, sum_values, compute_average]
    |> fn [c, s, a] => format_report(c, s, a)
```

9. Grammar Decisions

The four operators are lexed as distinct tokens: PIPE (|>), EPIPE (|?>), APIPE (|>>), and PPIPE (|>|). This avoids any ambiguity with the bitwise OR operator (|), which Lateralus also provides but uses as a prefix in pattern alternation rather than as an infix binary operator.

```
// Lateralus grammar (simplified BNF)
expr ::= atom
       | expr '|>'  expr        -- basic pipeline
       | expr '|?>' expr        -- error pipeline
       | expr '|>>' expr        -- async pipeline
       | expr '|>|' '[' expr (',' expr)* ']'  -- parallel fan-out
       | expr '+' expr          -- arithmetic, etc.
       | ...

-- All four pipeline forms are left-associative, same precedence level.
-- Pipeline binds less tightly than function application
-- but more tightly than assignment.
```

All four pipeline operators share the same precedence level and associate left-to-right. This means a mixed pipeline a |> f |?> g |>> h associates as ((a |> f) |?> g) |>> h, which is the intended reading: stages are applied in order left to right, with the operator type indicating what semantics apply at each step. The compiler's type-checker verifies that operator changes (e.g., from |> to |?>) are type-compatible at the transition point.

Parenthesization is permitted inside pipeline expressions, enabling sub-pipelines: a |> (b |?> c) |> d treats the error pipeline as a single stage within the outer basic pipeline. This is used in practice to compose reusable pipeline fragments: a library might expose a validated-fetch function implemented as |?> pipeline internally, which callers use in a basic |> pipeline after converting the result.

We considered and rejected the alternative of using a single operator |> that adapts its semantics based on the type of the right-hand function. This 'smart pipe' design (similar to the TC39 Smart-mix proposal) was rejected because it makes the semantics of a pipeline expression depend on the inferred types, which means the programmer cannot determine the pipeline's behavior from syntax alone. The four-operator design ensures that the semantics of each step is locally visible without reference to type-inference results.

The |>| parallel fan-out syntax requires a list literal on the right-hand side, not an arbitrary expression. This restriction is deliberate: it ensures the set of parallel stages is syntactically apparent at the use site and allows the compiler to determine at parse time how many parallel threads to spawn. A future extension may allow dynamic fan-out (where the list of functions is computed at runtime), but this is not in scope for the current design.

10. Type-System Integration

The pipeline operator integrates with Lateralus's Hindley-Milner type inference in a straightforward way for the basic and error variants: the type of the pipeline expression is inferred by unifying the output type of each stage with the input type of the next. The async and parallel variants require additional machinery: the async variant introduces an effect type Async, and the parallel variant introduces a product type for multiple return values.

```
 G |- e1 : T    G |- e2 : T -> U
---------------------------------- (T-Pipe)
      G |- e1 |> e2 : U

 G |- e1 : Result<T, E>    G |- e2 : T -> Result<U, E>
------------------------------------------------------- (T-EPipe)
         G |- e1 |?> e2 : Result<U, E>

 G |- e1 : Async<T>    G |- e2 : T -> Async<U>
------------------------------------------------ (T-APipe)
         G |- e1 |>> e2 : Async<U>

 G |- e : T    G |- fi : T -> Ui  for each i
---------------------------------------------- (T-PPipe)
   G |- e |>| [f1,...,fn] : (U1, ..., Un)
```

A key design decision is that Async is a type-level annotation, not a separate type entirely. An Async<T> value carries a T internally and is compatible with T-returning non-async functions via an automatic lifting rule. When a basic pipeline stage follows an async stage, the compiler inserts an implicit await to extract the T from the Async wrapper. This allows mixing async and sync stages without explicit annotation at every step.

```
// Type-system: mixing sync and async stages
// fetch_url : Url -> Async<Body>  (async)
// parse_html : Body -> Dom         (sync)
// extract_text : Dom -> String     (sync)

// The compiler automatically inserts await between async and sync stages:
let text : Async<String> =
    url
    |>> fetch_url       // Async stage
    |>  parse_html      // Sync stage: compiler inserts await(fetch_url result)
    |>  extract_text    // Sync stage

// Equivalent to:
let text = async {
    let body = await(fetch_url(url));
    parse_html(body) |> extract_text
}
```

The error type parameter E in the |?> variant is unified across all stages in the pipeline. This is a deliberate constraint: it forces the programmer to use a single, consistent error type throughout a pipeline, rather than having each stage introduce a different error type. In practice this is achieved by defining a sum type for all errors that can occur in a given pipeline domain (e.g., type AppError = ParseError | AuthError | DbError | NetworkError). The constraint can be relaxed using a future |?|> variant that maps error types automatically, but this is not in the 1.0 design.

Polymorphic pipeline stages are fully supported. A stage of type 'a -> 'b can appear in any pipeline where the inferred input type matches. This enables reusable pipeline fragments such as log (which passes its input through unchanged after logging it) or tee (which applies a side-effectful function and returns the input). These are not special-cased; they arise naturally from the polymorphism rules.

11. Error Propagation Model

The error propagation model for |?> pipelines is monadic in structure but designed to feel imperative in usage. The programmer writes a linear sequence of fallible stages and the language handles the branching. Under the hood, each |?> step checks whether the accumulated result is Ok or Err: if Err, subsequent stages are skipped and the error is forwarded; if Ok, the value is unwrapped and passed to the next stage.

```
// Error propagation: detailed example with error handling at the end
type DbError = ConnectionFailed | QueryFailed(String) | RowNotFound
type AppError = Db(DbError) | ParseError(String) | AuthError

fn fetch_user(id: UserId) : Result<User, AppError> =
    id
    |?> db_connect            // : UserId -> Result<Conn, AppError>
    |?> fn conn => query(conn, id)  // : Conn -> Result<Row, AppError>
    |?> parse_user_row        // : Row -> Result<User, AppError>

// Caller handles error at the end:
match fetch_user(user_id) {
    Ok(user)  => render_profile(user),
    Err(e)    => render_error(e),
}
```

The compiler emits efficient code for error pipelines. The naive implementation would check the Result tag at each stage boundary, emitting a branch that either calls the next stage or jumps to a shared error path. The compiler's error-short-circuit optimization hoists this branching structure to a single tag check at the beginning of each stage, which branch predictors handle efficiently because real-world pipelines succeed far more often than they fail.

For error pipelines that produce errors with attached context (e.g., a stack of error locations), Lateralus provides a |?>! variant that automatically wraps each stage's error in a context frame identifying the stage by name and source location. This enables rich error traces without requiring the programmer to manually add context at every step. The feature is opt-in to avoid overhead in performance-critical paths.

```
// Error context wrapping: automatic stage labeling
let result =
    request
    |?>! parse_request      // On error: wraps in Context("parse_request", loc, e)
    |?>! validate_auth      // On error: wraps in Context("validate_auth", loc, e)
    |?>! fetch_data         // On error: wraps in Context("fetch_data", loc, e)

// Error trace (if fetch_data fails):
// Error in pipeline at src/handler.lt:42:
//   at fetch_data (src/handler.lt:42:5)
//   at validate_auth (src/handler.lt:41:5)
//   at parse_request (src/handler.lt:40:5)
//   caused by: ConnectionFailed
```

The error type unification constraint is enforced by the type-checker at each |?> boundary. When a programmer uses functions with incompatible error types in a single pipeline, the error message names the specific stage where the mismatch occurred and shows both the expected and actual error type alongside a suggestion (usually: define a wrapper error type that includes both). This is the only case where the compiler's pipeline-awareness produces meaningfully better error messages than a desugaring approach could achieve.

12. Async Model

The async model in Lateralus is based on cooperative multitasking through a coroutine scheduler. Unlike Rust's async/await model (which generates state machines at compile time) or Go's goroutines (which use preemptive scheduling with green threads), Lateralus async is modeled as a monad over the scheduler effect. This gives it cleaner compositional properties while still generating efficient code via the IR-level async normalization pass.

The Async<T> type represents a computation that will eventually produce a T. It is parameterized only by its result type, not by error types (errors within async computations are carried by the Result type as usual). The |>> operator sequences two async stages by binding the first stage's future to the second stage's input. The scheduler is invoked at each bind point, allowing other tasks to run while the current task is suspended.

```
// Async model: scheduler integration
fn crawl(seed: Url) : Async<List<Page>> =
    seed
    |>> fetch_url             // yields to scheduler while fetching
    |>> parse_links           // synchronous, but within async context
    |>> fn links =>
            links
            |>| links.map(fn link => crawl(link))  // parallel async fan-out
            |>> flatten

// The scheduler sees each |>> boundary as a suspension point
// and can interleave multiple concurrent crawl() tasks.
```

Lateralus's async model differs from JavaScript's promise model in a critical way: Lateralus async computations are not eagerly started when created. A value of type Async<T> is a description of a computation, not a running computation. This 'cold' model (as opposed to JavaScript's 'hot' promises) means that async values can be stored, passed to functions, and composed without triggering execution. Execution begins only when the async value is passed to the scheduler's run function or awaited by a parent async pipeline.

The async pipeline integrates with the error pipeline through a combined Async<Result<T, E>> return type. This type appears so frequently in practice that Lateralus provides an alias AsyncResult<T, E> and a combined operator |?>> that sequences async-error stages with both short-circuit error propagation and scheduler suspension at each step.

```
// Combined async + error pipeline
fn process_request(req: Request) : AsyncResult<Response, AppError> =
    req
    |?>> authenticate        // Async<Result<AuthToken, AppError>>
    |?>> fn token =>
             token
             |?>> fetch_user_data    // parallel fetch
             |?>> fetch_permissions  // runs after fetch_user_data
    |?>> fn (data, perms) => authorize(data, perms)
    |?>> build_response
```

13. Parallel Fan-Out

The |>| parallel fan-out operator distributes a single value to multiple pipeline stages running concurrently. The implementation uses the work-stealing thread pool from Lateralus's runtime library. Each stage in the fan-out list is submitted as a task to the pool; the fan-out expression evaluates to a tuple of results collected after all tasks complete. The thread pool size defaults to the number of logical CPUs but can be configured at program startup.

```
// Parallel fan-out: benchmark example
// Sequential (baseline):
let stats = dataset
    |> compute_mean
    |> fn m => (m, compute_variance(dataset), compute_median(dataset))
// Problem: compute_variance and compute_median run after compute_mean

// Parallel fan-out (4x speedup on 4+ core machines):
let (mean, variance, median, mode) =
    dataset
    |>| [compute_mean, compute_variance, compute_median, compute_mode]
// All four run concurrently; result collected when all finish
```

The type rules for parallel fan-out require all stage functions to accept the same input type. When stage output types differ, the result is a heterogeneous tuple; the Lateralus type system encodes this as a product type with one component per stage. Pattern matching on the result (as in the example above) allows extracting each component with the appropriate type.

The parallel fan-out operator composes naturally with subsequent pipeline stages. After a fan-out, the programmer typically applies a merge function that combines the parallel results into a single value for further processing. This merge function is written as a regular lambda, keeping the pipeline style intact. The compiler emits efficient synchronization code: a join barrier waits for all parallel tasks before passing the tuple to the merge function.

```
// Fan-out composing with subsequent stages
let recommendation =
    user_id
    |>| [fetch_user_profile, fetch_purchase_history, fetch_browsing_data]
    |> fn (profile, history, browsing) =>
           compute_recommendations(profile, history, browsing)
    |> rank_recommendations
    |> take(10)

// The compiler emits:
// 1. Spawn 3 tasks: profile, history, browsing
// 2. Join barrier: wait for all 3
// 3. Call compute_recommendations with results
// 4. Call rank_recommendations
// 5. Call take(10)
```

An important correctness property of the parallel fan-out is that all stages observe the same value of the input. If the input contains mutable state (via Lateralus's controlled-mutation cells), the parallel stages may observe race conditions. Lateralus's type system prevents this: the input type must satisfy the Send bound (meaning it can be safely shared across threads), and mutable cell types do not satisfy Send. This gives a compile-time guarantee that parallel fan-out is data-race free.

14. Toolchain Implications

Retaining pipeline structure through compilation has significant implications beyond optimization. The language server protocol (LSP) implementation in Lateralus provides pipeline-specific hover information: hovering over a |> token shows the inferred input and output types of the stage, not just the type of the expression at the cursor. This makes navigating large pipelines significantly more convenient than in languages where the pipeline has been desugared.

The debugger integration is likewise pipeline-aware. Setting a breakpoint on a pipeline stage causes execution to pause before the stage's input is passed to the stage function. The debugger displays the stage name, the input value, and the inferred type of the expected output. Stepping forward executes the stage and displays the output. This contrasts sharply with debugging nested function calls, where the call stack reveals nothing about the programmer's intended data-flow structure.

```
// Debugger output (conceptual):
// Paused at pipeline stage 3 of 6 (validate_auth)
// ------------------------------------------------
// Input  : Parsed { method: POST, path: /api/users, token: "Bearer ..." }
// Type   : Parsed -> Result<AuthToken, AppError>
// Stage  : validate_auth [src/handler.lt:41]
//
// Previous stages:
//   1. parse_request  -> Ok(Parsed { ... })
//   2. rate_limit     -> Ok(Parsed { ... })
// >
// (s)tep, (c)ontinue, (p)rint input, (q)uit
```

Profile-guided optimization (PGO) in Lateralus uses pipeline stage boundaries as natural instrumentation points. The profiler records the time spent in each stage and the fraction of pipelines that short-circuit at each error stage. This data is used to guide inlining decisions (hot stages are preferentially inlined) and to adjust the error-short-circuit hoisting threshold.

The formatter (ltlfmt) understands pipeline structure and formats long pipelines with one stage per line, indented consistently. This contrasts with formatters for languages where pipelines have been desugared: a formatter operating on the desugared AST cannot reliably recover the pipeline structure and may produce badly formatted nested calls. Lateralus's formatter operates on the pre-desugaring CST and always produces idiomatic pipeline layout.

15. Performance Analysis

We measured the performance of Lateralus's four pipeline variants against equivalent implementations in F#, Elixir, OCaml, and JavaScript (Node.js) on our five benchmark programs. All measurements were taken on a 12-core AMD Ryzen 9 5900X at 3.7 GHz with 32 GB RAM, running Ubuntu 22.04. Each benchmark was run 100 times with 10 warmup iterations; we report median throughput in million operations per second (Mops/s).

```
Benchmark 1: JSON Processing Chain (1M records)
------------------------------------------------
Lateralus  |>   C99 backend:   847 Mops/s
Lateralus  |>   Python backend: 12 Mops/s
F#         |>   .NET 7:         203 Mops/s
Elixir     |>   BEAM:            38 Mops/s
OCaml      |>   native:         391 Mops/s
JavaScript |>   V8:              89 Mops/s

Benchmark 2: HTTP Request Pipeline (simulated, no I/O)
```

```
--------------------------------------------------
Lateralus  |?>  C99 backend:   712 Mops/s
F#         CE:                198 Mops/s
Elixir     with:               29 Mops/s
OCaml      let*:              344 Mops/s
JavaScript try/catch:          71 Mops/s
```

The performance advantage of Lateralus over F# and OCaml on the basic pipeline benchmark (847 vs. 203 vs. 391 Mops/s) is attributable primarily to stage fusion: the Lateralus compiler fuses the six stages of the JSON processing pipeline into two stages (parse + filter combined, map + sort + take combined) by recognizing adjacent list-traversing operations. This fusion eliminates four intermediate list allocations. F# and OCaml's desugaring-based approaches cannot perform this fusion reliably because the pipeline structure is not visible to the optimizer.

The error pipeline benchmark shows a smaller but consistent advantage over OCaml's let* style (712 vs. 344 Mops/s). The performance difference is due to two factors: first, Lateralus's error-short-circuit hoisting restructures the error check from per-stage branching to a single branch at pipeline entry when the compiler can prove (via type analysis) that the input is always Ok; second, Lateralus's Result representation uses a tagged integer rather than a heap-allocated discriminated union, saving one allocation per pipeline invocation.

```
Benchmark 3: Parallel Fan-Out (8-stage, 4 and 8 cores)
------------------------------------------------------
Lateralus  |>|  8 cores:   3,218 Mops/s  (3.8x sequential)
Lateralus  |>|  4 cores:   1,871 Mops/s  (2.2x sequential)
Lateralus  |>   seq. only:   847 Mops/s  (baseline)
F#         Async.Parallel: 1,102 Mops/s  (8 cores, 1.3x baseline)
Elixir     Task.async:     1,447 Mops/s  (8 cores, BEAM overhead)
OCaml      Domain:           891 Mops/s  (8 cores, 5.0 domains)

Benchmark 4: Async Streaming (10k URLs, network-simulated)
----------------------------------------------------------
Lateralus  |>>  work-stealing: 9,841 req/s
JavaScript async/await:        7,203 req/s
Elixir     Task.async_stream:  8,912 req/s
F#         Async.StartChild:   5,441 req/s
```

The parallel fan-out results are particularly noteworthy. Lateralus achieves 3.8x sequential throughput on 8 cores, compared to 1.3x for F# Async.Parallel and 1.05x for OCaml Domains on the same benchmark. The key factor is that Lateralus's |>| operator batches all task submissions in a single scheduler call, while F# and OCaml require per-task overhead for each parallel branch. The Lateralus work-stealing pool was tuned specifically for the small-task, high-fan-out pattern that |>| produces.

16. Related Work

The pipeline operator has deep roots in dataflow programming, shell scripting (the Unix pipe |), and functional programming (Backus's FP language, 1978). ML-family languages have used reverse-application as an idiom since the 1980s; the spelling |> was popularized by F# in the early 2000s. Our contribution is not the operator itself but the argument that it should be a first-class syntactic form.

The closest prior work to Lateralus's design is Koka (Leijen, 2014), which also retains effect information through its IR and uses it to drive optimization. Koka's row-typed effects are more general than Lateralus's Async/Err treatment but require more complex type annotations. We draw on Koka's insight that effect information should not be erased early in compilation but diverge in treating pipeline stages rather than effects as the primary unit of analysis.

Ropes (Boyapati et al., 2003) and Cyclone (Jim et al., 2002) explored region-based memory management for safe systems programming; our memory model (described in the companion ownership paper) is influenced by this work but integrated with pipeline stages as the natural region boundaries. Each pipeline stage may own its input allocation and transfer ownership to the next stage, eliminating unnecessary copies.

The decision tree approach to compiling pipeline fan-out is related to work on compiling parallel patterns (McCool et al., 2012, 'Structured Parallel Programming'). Lateralus's |>| operator is a restricted form of the 'map' pattern in that framework; the restriction to a statically known fan-out list allows compile-time scheduling decisions not available in the general map case.

The Hack-style TC39 pipeline proposal is related to the concept of 'holes' or 'topic variables' in concatenative languages (Forth, Joy, Factor). Our rejection of that style is documented in Section 5 above; a more detailed critique of the topic-variable approach can be found in Kudasov (2023), 'Against Implicit Topics in Pipeline Operators'.

17. Limitations and Future Work

The four-operator design has known limitations. First, mixed pipelines (using multiple operator variants in a single chain) require the programmer to be aware of the type-level implications at each transition. In practice, most pipelines use only one or two variants, but pathological cases can produce confusing type errors when the programmer mistakenly mixes variants that require incompatible types.

Second, the parallel fan-out operator |>| currently requires a literal list of functions; dynamic fan-out (where the list is computed at runtime) is not supported. This means that pipelines whose parallelism structure depends on runtime input cannot use the |>| idiom and must fall back to explicit task spawning. We plan to add a |>* (dynamic fan-out) operator in a future release, with appropriate bounds on the dynamic list's element type.

Third, the async model does not currently support back-pressure in streaming pipelines. A producer stage that emits values faster than a consumer stage can process them will cause unbounded buffering. We are designing a Stream<T> type and associated |>~ operator that provides back-pressure semantics, but this is not yet implemented.

Fourth, the |?> error type unification constraint, while disciplined, is sometimes too rigid for library design. A library function that returns Result<T, IoError> cannot be used directly in a pipeline that uses AppError without a wrapper. We are evaluating an automatic coercion rule that invokes a user-defined From trait to convert error types at pipeline boundaries.

Future work includes: (1) the |>~ streaming operator with back-pressure; (2) dynamic fan-out |>*; (3) automatic error-type coercion; (4) a pipeline inspector tool that visualizes the IR pipeline graph alongside profiling data; (5) integration with distributed tracing standards (OpenTelemetry) at the pipeline stage level; (6) formal verification of the stage-fusion optimization using Coq or Lean.

18. Conclusion

We have surveyed four existing pipeline operator implementations and identified three common failure patterns: error-propagation abandonment, async abandonment, and parallelism abandonment. All three failures stem from a single root cause: treating the pipeline operator as syntactic sugar rather than a first-class semantic form.

Lateralus addresses these failures through four pipeline operator variants (|>, |?>, |>>, |>|) that are first-class in the compiler's intermediate representation. This enables pipeline-specific optimizations (stage fusion, error-short-circuit hoisting, parallel task batching) that produce performance competitive

with hand-optimized code, as demonstrated by the benchmark results in Section 15.

The design is not without tradeoffs. The four-operator surface area is larger than any existing language's single-operator design. Programmers must learn when each variant is appropriate. We believe this cost is justified by the expressiveness gains: real-world programs require error handling, async I/O, and parallelism, and forcing programmers to abandon pipeline style whenever these are needed imposes a hidden cognitive cost far larger than learning four operators.

Lateralus is available at github.com/bad-antics/lateralus-lang. The benchmark suite used in this paper is available in the benchmarks/pipeline-survey directory of that repository. We encourage readers to run the benchmarks on their own hardware and contribute results for additional platforms.