Data-Flow Syntax: A Survey

Pipeline operators, dataflow graphs, and reactive streams across twelve languages

Lateralus Language

bad-antics · April 2024 · Lateralus Language Research

ABSTRACT Programmers have independently invented pipeline-like syntax in at least twelve mainstream languages over the past thirty years. These inventions span three distinct paradigms: operator-based pipelines (F#, Elixir, Hack, Lateralus), graph-based dataflow (LabVIEW, TensorFlow, Apache Beam), and reactive stream libraries (RxJava, ReactiveX, Kotlin Flow). We survey all three paradigms, classify them along five axes, and identify which aspects of the Lateralus pipeline model are genuinely novel versus which recapitulate known ideas. We conclude that first-class typed pipeline values with multiple operator variants is the one property absent from all prior work.

1. Scope and Classification Axes

We surveyed pipeline or dataflow mechanisms in: F# (|>), Elixir (|>), Hack (|>), OCaml (pipe_operators extension), Haskell ($, >>=), Rust (Iterator adapters), Julia (|>), Scala (for-comprehensions), Java (Streams API), LabVIEW (dataflow graph), TensorFlow (computation graph), and Kotlin Flow (reactive).

We classify each system along five axes:

- **First-class values**: can a pipeline be stored in a variable and passed to a function?
- **Multiple operator variants**: does the system distinguish total, error-propagating, async, and fan-out composition?
- **Type-level visibility**: does the type system know the pipeline is a pipeline, or does it desugar before type-checking?
- **Optimizer awareness**: can the compiler apply pipeline-specific optimizations (fusion, dead-stage elimination)?
- **Async integration**: is asynchronous composition a first-class concern, or an afterthought library?

2. Operator-Based Pipelines

2.1 F# |>

F# introduced the pipe-forward operator in 2005. It is purely syntactic: x |> f desugars to f x. There are no operator variants; error propagation requires the Result.bind function from the core library. The pipeline is not a first-class value. F# scores 1/5 on our classification axes (only operator precedence/readability).

2.2 Elixir |>

Elixir's pipe operator desugars to passing the left-hand expression as the first argument to the right-hand function. It is slightly more powerful than F#'s because the Elixir pattern-matching system makes error propagation with with blocks ergonomic, but the operator itself carries no type information and there are no variants. Score: 1/5.

2.3 Hack |>

The Hack language (PHP with types) added |> in 2021. Like F# and Elixir, it is syntactic sugar. The team also added $$ (placeholder syntax) for cases where the piped value is not the first argument. No error variant. Score: 1/5.

2.4 Haskell $ and >>=

Haskell's $ operator is right-associative function application with lower precedence: f $ g $ x reads right-to-left. It is the opposite of a pipeline in visual terms. The >>= (bind) operator is the monadic pipeline and is type-level visible, but each monadic chain is restricted to a single monad type. No fan-out operator. Score: 2/5 (type-level + monadic error).

3. Iterator Adapter Pipelines

3.1 Rust Iterator

Rust's Iterator trait provides a pipeline of lazy adapters: iter.filter(f).map(g).take(n).collect(). This is type-level visible (the chain is a typed expression), optimizer-aware (LLVM fuses the adapters), and zero-allocation for the chain itself. However, it is restricted to sequences: there is no native error-propagating adapter equivalent to |?>. filter_map handles optional values but not Result directly. Score: 3/5.

3.2 Java Streams API

Java 8 Streams provide a similar adapter chain: stream.filter(f).map(g).collect(toList()). The stream is a typed value, but it is not first-class in the sense of being storeable as a variable easily (without lambda capture). Error handling requires wrapping checked exceptions. Optimizer support is limited to within the stream implementation. Score: 2/5.

3.3 Kotlin Flow

Kotlin Flow extends the iterator model to asynchronous sequences. A Flow<T> is a cold, lazily-evaluated async stream. Operators include filter, map, flatMapConcat, and catch (for error handling). This is the closest prior art to Lateralus: typed, async-integrated, with error handling. However, flow operators are restricted to sequences; general function composition is not covered. Score: 4/5.

4. Dataflow Graph Systems

4.1 LabVIEW

LabVIEW's graphical programming model is a pure dataflow graph: nodes are functions, wires are typed connections. Execution order is determined by data availability, enabling automatic parallelism. Error clusters propagate through wires alongside data. This is the most 'first-class' pipeline model of any system we surveyed: the graph itself is a value that can be run, and sub-diagrams are composable. However, the graphical representation does not compose as a text syntax and has no type inference. Score: 4/5.

4.2 TensorFlow Computation Graph

TensorFlow 1.x used an explicit computation graph where operations are nodes and tensors are edges. The graph is a first-class value (tf.Graph), can be serialized, and is compiled by the XLA backend for optimization. TensorFlow 2.x shifted to eager execution, reducing graph-level first-classness. Error handling in graphs is separate from tensor flow. Score: 3/5 (TF1) / 2/5 (TF2).

4.3 Apache Beam

Apache Beam models distributed pipelines as a PCollection with PTransform stages. Pipelines are first-class values: a Pipeline object is constructed, transforms are applied to it, and then it is run on a specific runner. Error handling is done via dead-letter queues, not typed error propagation. Score: 3/5.

5. Reactive Stream Libraries

RxJava, RxJS, and ReactiveX model asynchronous event streams as Observable sequences with a rich set of combinators. The pipeline is a typed value (Observable<T>), combinators compose, and error handling is via the onError channel. Fan-out is supported via share() and publish(). These libraries score 4/5 but are restricted to event streams and carry significant runtime overhead (subscription graphs, scheduler allocation).

The key limitation of reactive libraries compared to Lateralus is that they are library-level abstractions: the compiler sees Observable as an opaque type and cannot fuse operators across the abstraction boundary. Lateralus achieves the same expressive power with compiler-level optimization.

6. Classification Table

Summarizing our survey:

System	First-class	Variants	Type-level	Optimizer	Async
F# |>	No	No	No	No	No
Elixir |>	No	No	No	No	No
Haskell >>=	No	Partial	Yes	No	No
Rust Iterator	No	No	Yes	Yes	No
Kotlin Flow	No	Partial	Yes	Partial	Yes
LabVIEW dataflow	Yes	Partial	No	Yes	Yes
Apache Beam	Yes	No	Partial	Yes	Yes
RxJava Observable	Yes	Partial	Yes	No	Yes
Lateralus	Yes	Yes	Yes	Yes	Yes

Lateralus is the only system to score 5/5. The critical differentiator is the combination of first-class pipeline values with multiple typed operator variants and compiler-level optimization. Every prior system achieves at most four of the five properties.

7. What Lateralus Borrows and What Is New

Lateralus borrows from prior art deliberately:

- The |> operator syntax from F# and Elixir.
- The typed error propagation model from Haskell's monadic bind.
- The iterator fusion strategy from Rust's compiler.
- The async integration model from Kotlin Flow.
- The first-class pipeline value concept from Apache Beam.

The genuinely novel contribution is the combination: a text-syntax language with multiple typed operator variants, first-class pipeline values, and compiler-level fusion. This combination did not exist in any prior system. The closest is Kotlin Flow + Rust Iterator, but the union of the two requires two separate APIs and two separate mental models.

8. Conclusion

Pipeline syntax has been independently re-invented at least twelve times. Each invention solves a subset of the problem; none solves it entirely. Lateralus's contribution is to identify the five properties that a complete solution requires and to implement all five in a single coherent language design.

The classification framework presented here is reusable: future pipeline designs can be evaluated against the same five axes and compared to this survey. We plan to maintain the survey as new pipeline proposals emerge in the literature and in production languages.