AMI: A Pipeline-Oriented Language:

Theory, Specification and Toolchain

Sam Caldwell

Asymmetric Effort, LLC.

5 Sept 2025

Dedicated to the girl

who listened to this idea in 1993.

Table of Contents

[Abstract 10](#_Toc209623870)

[1.0. Theory of Operation 11](#_Toc209623871)

[1.1. Pipeline-Oriented Programming (POP) 11](#_Toc209623872)

[1.1.1. POP and Event-Driven Programming 11](#_Toc209623873)

[1.1.2. Composable POP Segments 12](#_Toc209623874)

[1.1.3. Further Prior-Art Comparisons 13](#_Toc209623875)

[1.1.4. Problems in Declarative Programming 22](#_Toc209623876)

[1.1.5. Multiple Entrypoints 33](#_Toc209623877)

[1.1.6. Event Semantics 41](#_Toc209623878)

[1.1.7. Event Lifecycle 49](#_Toc209623879)

[1.1.8. Error Handling Pipelines 56](#_Toc209623880)

[1.2. Event Source Strategies 66](#_Toc209623881)

[1.2.1. Ingress Sources 66](#_Toc209623882)

[1.2.2. Source Capabilities and Constraints 68](#_Toc209623883)

[1.2.3. Source Isolation and Sandboxing 71](#_Toc209623884)

[1.2.4. Security and Trust Boundaries 73](#_Toc209623885)

[1.2.5. Design Trade-Offs 78](#_Toc209623886)

[1.3. Pipeline Composition 80](#_Toc209623887)

[1.3.1. Composition by Package Integration 80](#_Toc209623888)

[1.3.2. Event Typing and Compatibility 81](#_Toc209623889)

[1.3.3. Error Pipeline Integration 81](#_Toc209623890)

[1.3.4. Trust and Capability Propagation 82](#_Toc209623891)

[1.3.5. Package Versioning 82](#_Toc209623892)

[1.3.6. Operational Considerations 84](#_Toc209623893)

[1.3.7. Benefits and Limitations of Composition 85](#_Toc209623894)

[1.4. Concurrency, Parallelism and Event Loops 86](#_Toc209623895)

[1.4.1. Concurrency and Parallelism Distinguished 86](#_Toc209623896)

[1.4.2. Parallelism Presumption 87](#_Toc209623897)

[1.4.3. Event-Level Concurrency 88](#_Toc209623898)

[1.4.4. Worker Pools and Scheduling 98](#_Toc209623899)

[1.4.5. Error-Aware Scheduling 106](#_Toc209623900)

[1.4.6. Scheduling Policies and Guarantees 109](#_Toc209623901)

[1.5. Worker Functions 118](#_Toc209623902)

[1.5.1. Design Principles 119](#_Toc209623903)

[1.5.2. Execution Model and Sandbox 121](#_Toc209623904)

[1.5.3. Function Categories 125](#_Toc209623905)

[1.5.4. Purity, Immutability, and Controlled Mutability 127](#_Toc209623906)

[1.5.5. Function Reuse and Composition 131](#_Toc209623907)

[1.5.6. Worker Function Execution Context 141](#_Toc209623908)

[1.6. Observability and Telemetry 154](#_Toc209623909)

[1.6.1. Principles of Observability in POP 155](#_Toc209623910)

[1.6.2. Event-Level Telemetry 156](#_Toc209623911)

[1.6.3. Pipeline-Level Telemetry 158](#_Toc209623912)

[1.6.4. Node and Worker-Level Metrics 160](#_Toc209623913)

[1.6.5. Error and Anomaly Telemetry 162](#_Toc209623914)

[1.6.6. Longitudinal Analysis and Forecasting 165](#_Toc209623915)

[1.6.7. Integrating Telemetry with Development and Operations 167](#_Toc209623916)

[1.7. Event Typing and Schema Contracts 170](#_Toc209623917)

[1.7.1. Strong Typing of Events 170](#_Toc209623918)

[1.7.2. Schema Evolution and Compatibility 172](#_Toc209623919)

[1.7.3. Contracts Between Programs 174](#_Toc209623920)

[1.7.4. Type Safety vs. Flexibility Trade-Offs 175](#_Toc209623921)

[1.8. Program Structure and Composition 176](#_Toc209623922)

[1.8.1. Package and Namespace Organization 177](#_Toc209623923)

[1.8.2. Entry Points and Pipelines 179](#_Toc209623924)

[1.8.3. Error Pipelines as Structural Backstops 180](#_Toc209623925)

[1.8.4. Linking and Deployment 182](#_Toc209623926)

[1.9. Program-Level Semantics and Guarantees 184](#_Toc209623927)

[1.9.1. Safety Guarantees 184](#_Toc209623928)

[1.9.2. Delivery Guarantees 186](#_Toc209623929)

[1.9.3. Observability Guarantees 187](#_Toc209623930)

[1.9.4. Operational Guarantees and Limits 189](#_Toc209623931)

[1.9.5. Comparative Perspective 191](#_Toc209623932)

[1.9.6. Summary 193](#_Toc209623933)

[2.0. POP Programming in AMI 198](#_Toc209623934)

[2.1. Lexical Structure 199](#_Toc209623935)

[2.1.1. Source Text and Encoding 200](#_Toc209623936)

[2.1.2. Identifiers 200](#_Toc209623937)

[2.1.3. Keywords 201](#_Toc209623938)

[2.1.4. Pre-declared Identifiers and Reserved Words 202](#_Toc209623939)

[2.1.5. Literals 204](#_Toc209623940)

[2.1.6. Operators, Symbols, and Delimiters 205](#_Toc209623941)

[2.1.7. Comments and Whitespace 207](#_Toc209623942)

[2.1.8. The Special Underscore (**\_**) Identifier 207](#_Toc209623943)

[2.1.9. Package Identifiers 209](#_Toc209623944)

[2.1.10. Compiler Directives (Pragmas) 211](#_Toc209623945)

[2.2. Constructing Pipelines by Example 212](#_Toc209623946)

[2.2.1. Project Manifest (**ami.workspace**) 212](#_Toc209623947)

[2.2.2. Source File (**main.ami**) 216](#_Toc209623948)

[2.2.3. Package Declaration 220](#_Toc209623949)

[2.2.4. Node-Chained Pipeline Notation 223](#_Toc209623950)

[2.2.5. Ingress Node 224](#_Toc209623951)

[2.2.6. Worker Function (WorkerFuncSpec) Signature and Semantics 228](#_Toc209623952)

[2.2.7. Edges 231](#_Toc209623953)

[2.2.8. Transform Node 234](#_Toc209623954)

[2.2.9. FanOut Node 236](#_Toc209623955)

[2.2.10. Collect Node 240](#_Toc209623956)

[2.2.11. Egress Node 245](#_Toc209623957)

[2.2.12. Egress().get()Configuration Property Streams 246](#_Toc209623958)

[2.2.13. Worker Function Factory Pattern 248](#_Toc209623959)

[2.2.14. Node-State Tables 249](#_Toc209623960)

[2.3. Imperative Programming in AMI 253](#_Toc209623961)

[2.3.1. Global Scope, Defined 253](#_Toc209623962)

[2.3.2. Pointers and Addresses 255](#_Toc209623963)

[2.3.3. Function Declaration Forms 257](#_Toc209623964)

[2.3.4. Data Mutability 261](#_Toc209623965)

[2.3.5. Function Access to Node-State Table 266](#_Toc209623966)

[2.3.6. Concurrency 267](#_Toc209623967)

[2.3.7. Enumerated Types (enum) 270](#_Toc209623968)

[2.3.8. Struct Types (struct) 271](#_Toc209623969)

[2.3.9. Dynamic Containers (map, set, slices) 274](#_Toc209623970)

[2.4. Memory Management 278](#_Toc209623971)

[2.4.1. Allocation Domains 278](#_Toc209623972)

[2.4.2. Ownership & RAII 282](#_Toc209623973)

[2.4.3. Per-VM Memory Management 287](#_Toc209623974)

[3.0. AMI Toolchain 292](#_Toc209623975)

[Code Samples 293](#_Toc209623976)

[Tables 298](#_Toc209623977)

[Figures 300](#_Toc209623978)

[Bibliography 303](#_Toc209623979)

Abstract

Modern software services increasingly operate in environments where latency, throughput and composability are paramount. Pipeline-Oriented architectures decouple producers, consumers and transformers into composable graphs which process data as streams, allowing systems to react to asynchronous stimuli without blocking threads. AMI (Asynchronous Machine Interface) derives from the pipeline-oriented paradigm and functional programming to reify the event–stream abstraction as a first‑class programming model. This document provides a comprehensive introduction into pipeline-oriented programming (POP) and the AMI programming language. It starts with a theoretical discussion of event‑based programming, grounded in published literature, introduces the concepts of POP and the AMI language, dives into the specifics of the language and its tool chain then finishes with a discussion of POP in terms of test-driven development and program design.

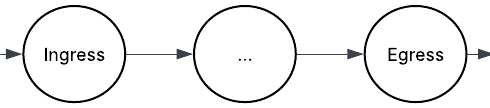
# Theory of Operation

## Pipeline-Oriented Programming (POP)

### POP and Event-Driven Programming

Pipeline-Oriented Programming (POP) is a paradigm in which a program is designed as a declarative graph of data processing “nodes” interconnected by FIFO/LIFO “edges.” Events are first-class, ingested from one or more heterogeneous and concurrent entrypoints, transmitted through edges to transformational information processing nodes and finally delivered to the external ecosystem through one or more concurrent egress nodes.

Figure 1: A simple POP Graph



Much like Yusupbekov and Mamatov, who envision an Event-Driven Programming (EDP) as a paradigm based on a first-class event, POP also envisions the event as first-class, though the pipeline itself is the focal point (Yusupbekov & Mamatov, 2016). The difference between their EDP and POP is the declarative pipeline, a data flow roadmap—complete, testable and even reusable. Yusupbekov and Mamatov note that an event‑driven application typically consists of three mandatory elements:

1. A source that produces events,

2. A channel or listener through which events are propagated, and

3. An event handler that processes the event.

This sounds similar to POP. But POP defines the program as having—

1. One or more asynchronous entrypoints,
2. Zero or more independent data transformation and routing paths, and
3. One or more outputs.

### Composable POP Segments

Figure 2: Composable POP Pipeline Segments

A diagram of a pipeline

AI-generated content may be incorrect.

Pipeline-Oriented Programming (POP) is concurrent and event-driven by nature, declarative at its highest level and composable. This moves the programmer further from the von Neumann style, as Backus proposed and toward a new distributed programming future (Backus, 1978). A pipeline may be composed of a simple ingress and egress node as a form of the famous “hello world” program, or it may be composed of many ingress nodes feeding a web of interdependent data transformation nodes before ultimately rendering one or many output streams to its consumer. POP envisions that these pipelines may exist on a single computer or they may span many computer systems. But each pipeline may be decomposed into individual testable pipeline segments. These testable pipeline segments make POP a natively test-driven development paradigm.

### Further Prior-Art Comparisons

#### Functional Programming

Pipeline-Oriented Programming (POP) shares important intellectual roots with functional programming (FP), particularly in its commitment to immutability, referential transparency, and compositional reasoning (Scott, 2016). In both paradigms, programs are constructed from small, deterministic units that can be combined into larger systems. This alignment reflects the call for liberating programming from the von Neumann bottleneck through algebraic forms of composition (Backus, 1978).

Figure 3: POP Data Pipeline

A diagram of a diagram

AI-generated content may be incorrect.

Despite these shared foundations, POP diverges significantly from FP in its execution model. Whereas FP organizes computation around the evaluation of pure expressions, POP organizes computation around the **flow of events through explicit pipelines**. Nodes in a POP graph correspond to processing units that transform, branch, or merge event streams. Unlike FP, which abstracts away evaluation order (Abelson, Sussman, & Sussman, 1996, p. 352), POP makes event sequencing and delivery strategies (e.g., FIFO, LIFO, backpressure policies) explicit parts of the language.

State management further illustrates the distinction. FP typically discourages mutable state altogether, relying on recursion and higher-order functions to achieve iteration and accumulation (Scott, 2016, pp. 537-590). POP, by contrast, enforces **immutability at pipeline boundaries**—ensuring that events cannot be altered once emitted—and enforces immutability by default elsewhere while allowing nodes to maintain internal, scoped state for tasks such as caching, aggregation, or metrics. This controlled allowance of state enables POP to support real-time and distributed systems more naturally than pure FP.

Finally, compositionality manifests differently in the two paradigms. FP composes functions into larger expressions; POP composes nodes into reusable pipeline segments that can be validated, tested, and scaled independently. Thus, POP adapts FP’s algebraic spirit but applies it to the orchestration of event-driven graphs rather than the manipulation of expressions.

#### Comparing the Actor Model

The Actor Model (Agha, 1986) is often cited as a canonical framework for reasoning about concurrency. Actors encapsulate mutable state, receive messages asynchronously, and may respond by altering their internal state, sending new messages, or creating additional actors. This model inspired many distributed systems and languages because it abstracts away the complexities of threads and locks.

Pipeline-Oriented Programming (POP) shares the Actor Model’s reliance on asynchronous message passing but adopts a more **declarative and deterministic structure**. In POP, programs are defined as **fixed graphs of nodes** rather than dynamic collections of actors. This distinction has several consequences:

* **Graph immutability versus dynamic spawning**: In the Actor Model, the structure of the system evolves at runtime as new actors are created. In POP, pipelines are declared in advance and remain structurally fixed, allowing for compile-time reasoning, optimization, and verification.
* **State handling**: Actors maintain private mutable state that evolves over time. POP enforces **immutability at pipeline boundaries**; nodes may hold local, temporary state for computation, but events themselves cannot be modified once emitted. This reduces nondeterminism and eases distributed reasoning.
* **Determinism and analysis**: Because POP pipelines are static graphs, they are amenable to **formal analysis, capacity planning, and static verification** in ways that actor networks are not. The Actor Model’s flexibility offers expressiveness but complicates static guarantees.
* **Error handling**: While Actor systems rely heavily on supervision hierarchies and restart strategies, POP introduces explicit **ErrorPipelines** to channel and process failures as first-class citizens in the graph.

#### Object-Oriented Programming

Object-Oriented Programming (OOP) has shaped modern software development through its core abstractions of **classes, objects, encapsulation, and inheritance** (Scott, 2016). In OOP, developers freely define classes, instantiate objects, and organize programs around mutable state and method invocation. These features have proven practical across domains, from systems programming to application frameworks (Weiskamp, Heiny, & Flamig, 1991; Stroustrup, 2003).

Pipeline-Oriented Programming (POP) appears to share some similarities with OOP. Both paradigms promote **modularity**—OOP by encapsulating behavior and state within objects, and POP by encapsulating event-processing logic within nodes. POP’s chained-node grammar even resembles OOP’s method chaining, where the dot (.) operator connects components. Yet the semantics of this connection diverge sharply. In OOP, the dot denotes a method call that frequently mutates internal state. In POP, the dot denotes an **abstract edge** in a declarative pipeline, representing event flow rather than control flow.

POP also restricts what OOP leaves unconstrained. OOP permits **arbitrary class hierarchies, inheritance, and polymorphism**, allowing developers to construct dynamic object graphs at runtime. POP, by contrast, limits its vocabulary to **five node types—**Ingress**(),** Transform()**,** FanOut()**,** Collect()**, and** Egress()—and requires pipelines to be declared statically. This design choice sacrifices flexibility for **analyzability and static verification**, enabling compile-time reasoning about flow, capacity, and concurrency.

State management further differentiates the paradigms. OOP revolves around mutable object state, using encapsulation to impose discipline on its manipulation (Stroustrup, 2003). POP enforces **immutability across node boundaries**: once an event is emitted, it cannot be modified. Nodes may hold scoped state for aggregation or caching, but such state never escapes into the event flow. This principle makes POP pipelines inherently safer for concurrency, avoiding the synchronization complexities that OOP typically requires external mechanisms—such as threads or locks—to manage.

In summary, OOP and POP both pursue modular design and composability, but from very different foundations. OOP achieves flexibility through classes, inheritance, and mutable state, while POP achieves reliability and analyzability by restricting itself to a fixed set of nodes, immutable event flows, and declarative pipeline graphs. These constraints yield strong guarantees of **concurrency safety, determinism, and formal analyzability** in event-driven systems.

#### Dataflow Programming

Dataflow programming represents one of the earliest alternatives to the von Neumann model, introduced in the 1970s as an attempt to express computation in terms of **graphs of operations linked by data dependencies** rather than sequential instruction streams. Dennis and Misunas (1975) describe a “dataflow processor” in which nodes execute as soon as all required inputs are available, with tokens of data flowing along edges to trigger further computation (Dennis & Misunas, 1975). This approach contrasts sharply with imperative and object-oriented paradigms, which rely on explicit control flow and shared mutable state (Ray & Kumar, 2007).

Pipeline-Oriented Programming (POP) is closely related to the dataflow lineage but introduces refinements for modern event-driven and distributed systems. Both paradigms express computation as graphs, but while classical dataflow models emphasized firing rules and token availability, POP focuses on **event streams** and **typed edges**. POP inherits dataflow’s natural support for concurrency—since independent nodes can operate simultaneously—but strengthens the model with additional constraints, such as **immutability across pipeline boundaries and explicit backpressure strategies** to regulate load.

Another important distinction lies in **graph mutability**. Classical dataflow systems often allowed highly dynamic graph construction, enabling fine-grained parallelism but complicating analyzability. POP enforces **statically declared pipeline graphs**, where cycles are permitted only when accompanied by explicit anti-deadlock strategies. This design allows POP to retain the expressive power of dataflow while ensuring pipelines remain analyzable and predictable at compile time.

In this sense, POP can be viewed as a pragmatic evolution of the dataflow paradigm: adopting its **graphical execution semantics and inherent concurrency**, while adapting them to the needs of **event-driven programming, distributed deployment, and safety guarantees** in modern software systems.

#### Staged Event-Driven Architecture (SEDA)

The Staged Event-Driven Architecture (SEDA) was proposed as a scalable design pattern for internet services, decomposing applications into a series of stages connected by explicit queues (Welsh, Culler, & Brewer, 2001). Each stage is responsible for a portion of the computation and communicates with others by passing events through queues. This design isolates failures, prevents overload by applying backpressure at queues, and simplifies reasoning about concurrency compared to monolithic event loops.

Superficially, SEDA resembles Pipeline-Oriented Programming (POP), since both paradigms decompose execution into modular components connected by event flow. However, POP departs in three critical ways. First, where SEDA stages are general-purpose and queues are runtime constructs, POP defines a restricted set of node types (Ingress, Transform, FanOut, Collect, Egress) with semantics analyzable at compile time. Second, while SEDA emphasizes elasticity through runtime scheduling policies, POP achieves elasticity declaratively by binding backpressure and concurrency strategies directly into pipeline graphs. Finally, SEDA’s queues act as untyped conduits between stages, whereas POP enforces strong typing of Event<T> objects and supports compile-time verification of all edges.

These differences highlight POP’s focus on **formal analyzability and type safety**, in contrast to SEDA’s emphasis on **operational robustness**. While SEDA provided an important stepping stone away from ad hoc event-driven designs, POP advances the paradigm by embedding concurrency, typing, and fault handling into the language itself.

#### Comparative Summary

Pipeline-Oriented Programming (POP) can be situated within a broad lineage of programming paradigms. Like functional programming, it emphasizes immutability and stateless transformations, though POP restricts side effects more rigorously by limiting all I/O to ingress and egress nodes. Compared to the actor model, POP similarly decomposes computation into isolated units communicating via immutable messages, but it departs by making these units declarative nodes within a statically analyzable graph rather than arbitrary actors instantiated at runtime. In relation to object-oriented programming (OOP), POP shares a syntactic resemblance in its use of chained node expressions, yet diverges fundamentally by constraining the class vocabulary to a fixed set of node types and enforcing immutability across node boundaries. Dataflow programming provides another close antecedent: POP inherits its graph-based execution semantics, but tightens guarantees by coupling backpressure, delivery semantics, and typing rules directly to the pipeline grammar.

The Staged Event-Driven Architecture (SEDA) is also relevant prior art, decomposing computation into staged queues that isolate failures and regulate load (Welsh, Culler, & Brewer, 2001). POP draws on SEDA’s insight that event-driven systems benefit from explicit staging and backpressure but advances the model by embedding analyzability, typing, and error handling into the language. Unlike SEDA, which treats stages and queues as runtime constructs, POP enforces its concurrency and safety properties at compile time.

Taken together, these comparisons illustrate how POP both builds on and departs from earlier paradigms. By combining immutability, declarative graph structure, strong typing, and constrained I/O boundaries, POP unifies lessons from multiple traditions while offering a distinct design space optimized for analyzability, concurrency safety, and operational observability.

### Problems in Declarative Programming

Declarative pipeline graphs in Pipeline-Oriented Programming (POP) provide clarity and analyzability, but they also surface well-known challenges from the broader declarative programming tradition. Breaking these challenges down into focused subsections provides a clearer discussion:

#### Loops and Cycles

One of the most prominent challenges in declarative programming arises from the handling of **loops and cycles**. Declarative specifications often make it easy to describe cyclic relationships but difficult to reason about their runtime implications. In pipeline graphs, cycles can express useful patterns—such as retries, feedback, or iterative aggregation—but they also risk introducing infinite recursion, deadlock, or unbounded resource consumption if not properly constrained.

A classic example comes from logic programming. Called the “marriage problem,” it demonstrates how a Prolog program can enter an infinite loop (Abelson, Sussman, & Sussman, 1996). It starts with the declarative statement—

|  |
| --- |
| (assert! (married Minney Mickey)) |

Code 1: Prolog Marriage Problem Fact

The user then asks the question—

|  |
| --- |
| (married Mickey ?who) |

Code 2: Prolog Marriage Problem Query

Since the system knows Minney is married to Mickey, but it has no rule to know that this implies Mickey is married to Minney. But when the rules are defined as follows, things break down further:

|  |
| --- |
| (assert! ( rule( married ?x ?y ) ( married ?y ?x ) ) |

Code 3: Prolog Marriage Problem Infinite Loop

Now, the program enters an infinite loop when the question of Mickey and Minney’s marital status is asked. This technology creates the equivalent of Abbott and Costello’s “Who’s on First?” and illustrates how a seemingly intuitive declarative specification can yield a non-terminating evaluation unless carefully bounded (Abelson, Sussman, & Sussman, 1996).

The challenge is not unique to POP. Dataflow programming, which represents programs as graphs of operations and data dependencies, has long faced similar difficulties. Dennis and Misunas (1975) observed that unregulated feedback loops in dataflow architectures could cause tokens to circulate indefinitely, leading to nondeterministic execution or resource exhaustion. More recent work has highlighted that asynchronous dataflow programs can exhibit subtle non-termination behaviors unless cycles are paired with well-defined scheduling and validation rules (Lin, Gancher, & Parno, 2023). These findings underscore that cycles in declarative systems require explicit **safeguards and validation mechanisms** to ensure predictable execution.

POP’s design acknowledges this risk directly. Pipelines may include loops, but only when accompanied by **explicit anti-deadlock strategies**. For example, a Collect() node may feed back into an upstream Transform() node, but the loop must be annotated with bounded queue capacities, breaker conditions, or timeout-based gates. These mechanisms prevent pipelines from devolving into the equivalent of Prolog’s infinite marriage inference. In this way, POP preserves the expressiveness of cycles while constraining their behavior to ensure **safety, analyzability, and predictable execution.**

#### Cross-Package References

Declarative pipelines often span multiple packages, creating references across package boundaries. This modularity increases reuse but also raises challenges around dependency management, version compatibility, and semantic clarity. To address these challenges, POP enforces several structural rules:

1. **Single-package definition**: An entire pipeline segment—from Ingress() to Egress()—must be defined within a single package namespace.[[1]](#footnote-1)
2. **Cross-package invocation**: A pipeline segment may be invoked by another segment across a package boundary.
3. **Nested invocation**: A pipeline segment may itself invoke other pipeline segments defined in different packages.

At compile time, each invocation of a pipeline segment is instantiated as a **separate logical entity** when necessary to enforce type safety. This prevents type collisions across package boundaries and ensures that event types remain distinct. The result is that a declarative pipeline may span many packages, but the compiler guarantees integrity by treating each segment instance independently.

The need for such constraints is supported by studies of packaging ecosystems. Research shows that focal packages in software ecosystems can become brittle points of failure: changes or removals cascade across downstream dependencies in unpredictable ways (Qi, 2024). Similarly, transitive dependencies accumulate rapidly, creating hidden couplings that complicate reasoning about software behavior (Decan, Mens, & Grosjean, 2017). POP’s compile-time instantiation rules address these concerns directly by **isolating pipeline segments** and **enforcing type-safe boundaries** between pipeline segments. [[2]](#footnote-2)

Thus, POP combines the **reuse benefits of modularity** with safeguards that reduce the risks of fragile dependency chains. Declarative pipelines remain analyzable and predictable, even as they cross package boundaries, because the compiler enforces distinct instantiation and type integrity for each reference.

#### Static vs. Dynamic Ambiguities

Declarative pipeline graphs are attractive because they promise compile-time analyzability, predictability, and safety. However, as demonstrated in the dataflow literature, real systems often require dynamicbehaviors such as conditional branching, runtime variation, or graph mutation (Ha & Lee, 1997). These features introduce ambiguities that complicate static analysis and undermine some of the guarantees of declarativity (Lee & Bier, Architectures for statically scheduled dataflow, 1990).

A key advantage of purely static graphs is that they support compile-time reasoning and optimization. Lee and Bier (1990) showed that when dataflow graphs are *data independent*—meaning node firing does not depend on runtime values—they can be scheduled statically with high efficiency and low runtime overhead. This determinism enables strong guarantees and predictable execution.

In contrast, many real systems cannot avoid dynamic constructs. Ha and Lee (1997) demonstrated that conditional edges, data-dependent iteration, and recursion can be incorporated into dataflow programs, but only by making assumptions based on profiles or statistical models of execution. Such assumptions introduce fragility: if runtime behavior diverges from profiles, the guarantees made at compile time may no longer hold. Research further highlights that asynchronous dataflow programs often exhibit nondeterministic or non-terminating behaviors when dynamic constructs are not tightly controlled, emphasizing the importance of formal validation strategies (Lin, Gancher, & Parno, 2023).

POP addresses this tension by preserving a static pipeline structure while allowing localized dynamic behavior within nodes. Specifically, conditional logic is expressed through Transform() and FanOut() nodes. Transform() nodes contain stateless worker functions that execute in a restricted virtual machine with no I/O or shared mutable state. FanOut() nodes declaratively define multiple static paths with routing predicates that direct Event<T> objects to exactly one downstream branch at runtime. The compiler enforces exhaustiveness and exclusivity of FanOut() rules wherever possible, and any runtime mismatch results in the event being wrapped as an Error<E> and redirected through the error-handling pipeline.

We accept a deliberate trade-off: graph-level declarativity enables strong compile-time reasoning, while node-local flexibility reduces some global optimization opportunities. The compiler prioritizes graph-level optimizations (e.g., scheduling, capacity planning, pipeline fusion where legal) while applying conventional intra-node optimizations to worker code (e.g., inlining, constant folding). Because worker functions are stateless and restricted, these optimizations are safe but less powerful than those available in purely declarative systems.

Structural correctness is established at compile time. Observability mechanisms such as the strongly typed Event<T> object container which allows traceability from end to end along with constraints such as capacity limits and timeouts provide runtime assurance for throughput and backpressure without weakening compile-time guarantees. In fact, an implementing language can use Event<T> metadata to make runtime optimizations.

By maintaining a strict declarative pipeline structure and allowing only carefully bounded dynamic code within nodes, POP balances the competing interests of flexibility and analyzability. This design minimizes semantic drift while preserving determinism, concurrency safety, and predictable system behavior. (Chukhman, Jiao, Salem, & Bhattacharyya, 2016)

#### Validation Complexity

Declarative pipeline graphs are concise, but that brevity can conceal under-specification. Edges may connect nodes without defining buffer policies, ordering guarantees, or backpressure limits. Without sufficient detail, the declarative model risks drifting away from the realities of execution. This tension is well documented in dataflow research. Dataflow applications frequently exhibit performance or correctness mismatches between specifications and runtime behavior, requiring post-hoc instrumentation to detect failures (Chukhman, Jiao, Salem, & Bhattacharyya, 2016). Similarly, we find evidence that formal verification of dataflow compilers was necessary to ensure compiled systems behave consistently with their declarative semantics (Bourke, Brun, & Pouzet, 2019).

In POP, this problem is addressed by I embedding validation hooks directly into the runtime model. Every Event<T> and Error<E> object carries tracing and metadata, providing visibility into pipeline execution without the need for bolted-on profilers. These metadata support runtime optimizations, throughput monitoring, and end-to-end tracing of event lifecycles.[[3]](#footnote-3) When validation fails—such as a FanOut() predicate mismatch or an invalid loop iteration—the system automatically wraps the failing object in an Error<E> and routes it through the error pipeline, ensuring that runtime violations are surfaced as structured, analyzable events.

Compile-time checks further reduce validation complexity. The POP compiler enforces FanOut()coverage and exclusivity, type integrity across pipeline segments, and Collect() loop constraints. This guarantees that the pipeline graph is structurally sound before execution. At runtime, observability mechanisms—timeouts, queue capacity limits, telemetry streams—provide runtimevalidation. As Charity Majors observes, “You’ll never know — not really — what the code you wrote does just by reading it. The only way to be sure is by instrumenting your code and watching real users run it in production” (Majors, In Praise of “Normal” Engineers , 2025). POP makes this principle intrinsic: observability is not an afterthought but a first-class feature of the language.

By combining compile-time enforcement, built-in runtime metadata, and structured error handling, POP reduces the gap between declarative specification and operational reality. Validation complexity remains a challenge, but in POP it becomes a managed property of the system, not a latent liability.

#### Debugging and Traceability

Debugging declarative systems has historically been difficult because specifications describe *what* the system should do without showing *how* execution unfolds. Observability is an afterthought punted to an operations team after the fact. Failures such as dropped events, backpressure stalls, or deadlocks can be hard to trace when the declarative syntax provides no direct handle into runtime behavior. Researchers in dataflow validation note that mismatches between specification and implementation are often subtle and only emerge under load (Chukhman, Jiao, Salem, & Bhattacharyya, 2016). Without built-in traceability, debugging becomes an external exercise in profiling, log mining, and inference.

POP reduces this complexity by implementing observability (e.g. tracing) as a language/paradigm feature at the core event model. Every Event<T> and Error<E> object carries metadata for lineage, timestamps, causal context, and error provenance. This ensures that even as pipelines scale across packages and nodes, each event remains identifiable and traceable through its entire lifecycle. Errors are not silent failures but structured objects (Error<E>) that move through the error-handling pipeline, preserving context for analysis.

Operational observability reinforces this. In POP, this principle is realized by design: instrumentation is intrinsic, not an afterthought. Telemetry, timeouts, and capacity monitoring allow developers to correlate declarative intent with runtime traces, reducing the cognitive gap between specification and observed behavior.

By combining compile-time validation with event-level metadata and structured error pipelines, POP makes debugging and traceability part of the language itself. Developers do not need to retrofit tracing frameworks or guess at runtime state: the pipeline graph, event metadata, and observability hooks form a closed loop of specification, execution, and introspection

### Multiple Entrypoints

A diagram of a diagram

AI-generated content may be incorrect.

Figure 4: Multiple Entrypoint POP Program

POP pipelines are not limited to a single root. Instead, a declarative pipeline may define multiple Ingress() nodes as independent entrypoints. This reflects the realities of distributed systems and data processing frameworks, where multiple data sources often need to be processed concurrently (Dennis & Misunas, 1975; Borkar, Carey, Grover, Onose, & Vernica, 2010; Dean & Ghemawat, MapReduce: Simplified data processing on large clusters, 2008).

Multiple entrypoints introduce additional complexity for analysis, scheduling, and debugging, but POP’s design ensures analyzability and observability through its type system, compile-time validation, and embedded metadata.

#### Structural Rules

POP supports multiple entrypoints by allowing pipelines to declare more than one Ingress() **node**. Each entrypoint is defined as the execution head of a dynamically scalable fleet of **worker functions in a given** Ingress() **node**, expressed in LLVM, and permitted to perform I/O operations only within the constraints of a defined capabilities **list**. This makes entrypoints the exclusive gateways where external data can enter a pipeline, preserving analyzability and safety elsewhere in the graph.

Figure 5: Ingress Node as Entrypoint

A diagram of a work flow

AI-generated content may be incorrect.

To ensure scalability and concurrency, each entrypoint is fronted by a **load-balancing head**. This dispatcher spreads incoming Event<T> objects across a fleet of dynamically scalable workers, ensuring that throughput can adapt elastically to load while maintaining type safety and immutability guarantees.

At compile time, the POP compiler instantiates each entrypoint independently, preserving **type isolation, attributes** and ensuring that pipeline segments can be analyzed as distinct units even when composed across packages. These constraints mirror principles established in early dataflow systems, which required explicit firing rules and token-based synchronization to preserve analyzability in graphs with multiple sources (Dennis & Misunas, 1975). More recently, large-scale declarative frameworks such as Dryad and MapReduce adopted similar patterns, supporting multiple data sources while requiring static contracts to maintain correctness (Dean & Ghemawat, MapReduce: Simplified data processing on large clusters, 2008; Borkar, Carey, Grover, Onose, & Vernica, 2010).

By enforcing these structural rules, POP makes multiple entrypoints tractable: each Ingress() is tightly scoped, observable, and analyzable, while still supporting the elastic scalability needed in distributed event-driven systems.

#### Concurrency and Safety

Multiple entrypoints imply that a pipeline may receive **independent streams of events** concurrently. Left unmanaged, this could create nondeterministic race conditions, resource exhaustion, or backpressure failures. POP addresses these risks with three layers of protection:

1. **Immutable Events.** All Event<T> objects are immutable once created, ensuring that concurrent workers cannot corrupt shared state. This approach follows the well-established practice in dataflow systems of treating tokens as immutable carriers of data (Dennis & Misunas, 1975) and Functional Programming (Scott, 2016, pp. 535,et seq.).
2. **Load-Balanced Ingress Fleets.** Each Ingress() node is fronted by a load-balancing head that dispatches incoming Event<T> across a dynamically scalable fleet of workers. This allows pipelines to elastically absorb bursts of input without introducing contention for shared resources.
3. **Backpressure and Resource Limits.** Worker pools are bounded, and queue depths are explicitly declared in pipeline specifications. When downstream capacity is exceeded, backpressure propagates upstream, enforcing stability and preventing cascading failures. Declarative frameworks such as Dryad and modern streaming systems (Borkar, Carey, Grover, Onose, & Vernica, 2010) rely on similar backpressure mechanisms to maintain correctness under load.

Concurrency safety also depends on consistenteventidentification. Each Ingress() node worker function transparently assigns a unique EventId to every incoming event, composed of:

* The Ingress() node’s program-unique identity, assigned by the compiler,
* The runtime-generated worker number within the Ingress() fleet that processed the event, and
* A 64-bit unsigned sequential integer that increments for each event processed by that worker during a given runtime.

An EventId is guaranteed to be unique only for a given program runtime. Sequential numbering resets when a program restarts, and two independent executions of the same build share the same Ingress node identities, but the worker node identity and event identity sequence numbers vary. This scoping ensures that an EventId is sufficient for runtimeobservability,debugging, and trace correlation which do not require globally unique identifiers in this context.

Together, immutability, bounded concurrency, load-balanced ingress fleets, and scoped the EventId allow Pipeline-Oriented Programming to handle multiple entrypoints without sacrificing determinism or safety.

#### **Observability and Debugging**

Debugging declarative systems has historically been difficult because specifications describe what should happen without exposing how execution unfolds. In POP, observability is not an afterthought but an intrinsic property of the event model. Every Event<T> object carries metadata for traceability, including its EventId, timestamps, lineage, and causal context.

The EventId provides the backbone of traceability. Each event is identified by a tuple consisting of:

* The Ingress() node’s identity, assigned by the compiler and stable across builds.
* The worker number within the ingress fleet that processed the event.
* A 64-bit unsigned sequential integer that increments for each event accepted by that worker during a given runtime.

This identifier scheme ensures uniqueness withinagivenprogramexecution, while maintaining determinism across builds. Sequential numbering resets on program restart, and two copies of the same build will share Ingress node identities. Thus, an EventId is designed for runtime trace correlation and debugging, not global uniqueness.

Errors in POP are treated as first-class events. An error does not simply signal failure; it encapsulates the original Event<T> within an Error<Event<T>> object (also represented as Event<E>). This wrapper preserves the complete event context and augments it with error-specific metadata: failure cause, stack trace, and relevant diagnostics. By routing errors through a dedicated error-handling pipeline, POP ensures that debugging information is both structured and analyzable.

This approach is aligned with industry best practices in large-scale distributed systems. The Google *Site Reliability Engineering* book emphasizes that monitoring and observability depend on structured logs, metrics, and tracing to make distributed execution visible and diagnosable (Beyer, Jones, Petoff, & Murphy, 2016). The *Monitoring Distributed Systems* extension highlights that observability must capture not only the “happy path” but also unexpected divergences in system behavior (Bjork, Burns, Fong-Jones, & Hochstein, 2020, pp. 231-266). Similarly, Krabbe (2019) stresses that effective instrumentation balances completeness with clarity, recording high-signal contextual data without overwhelming operators with noise (Krabbe, 2019). POP’s choice to embed metadata directly into events reflects these principles: instrumentation is automatic, scoped to each event, and sufficient for both performance monitoring and debugging.

By combining a scoped EventId, metadata-rich Event<T> objects, and structured error handling, POP makes debugging and traceability part of the language itself. This design not only reduces the cognitive burden of understanding distributed pipelines but also sets the stage for a deeper discussion of **event semantics**. In the next section, we examine how immutability, lineage, and metadata converge to define the guarantees and behaviors of POP events, forming the foundation for analyzability and observability at scale.

#### **Benefits and Trade-offs**

Allowing multiple entrypoints provides POP with significant benefits in terms of expressiveness, modularity, and scalability. Pipelines can integrate heterogeneous sources—such as APIs, file streams, and sensors—into a single declarative graph. Entry points are statically declared, analyzable at compile time, and scalable at runtime through load-balanced Ingress() fleets. This makes POP well-suited for modern distributed systems, where concurrency and elasticity are essential.

The trade-offs are equally important. Multiple ingress points increase the complexity of validation, scheduling, and debugging. Concurrency risks emerge when independent streams interact, and operational overhead grows as the number of entrypoints scales. Without strong constraints, such designs could undermine the declarative guarantees POP aims to provide.

POP mitigates these risks through a layered approach. Immutableevents ensure concurrency safety, while Ingress()-assigned EventId values provide traceability across entrypoints. Error<Event<T>> objects encapsulate failures in structured form, and **observability metadata** embedded in every event ensures that pipeline behavior remains visible. Compiler-enforced structural rules and runtime resource limits further contain the complexity.

In sum, the benefits of multiple entrypoints outweigh the trade-offs, provided they are disciplined by POP’s design principles. The result is a programming paradigm that combines the flexibility of multi-source pipelines with the analyzability, safety, and observability of a declarative language.

### Event Semantics

Events are the fundamental abstraction in Pipeline-Oriented Programming (POP). Every pipeline is defined in terms of Event<T> objects flowing between nodes, through edges, with semantics that guarantee analyzability, concurrency safety, and observability. Understanding event semantics is essential because they unify the declarative, imperative and operational dimensions of the paradigm.

To capture these guarantees, POP defines event semantics along several axes:

* **Immutability**, ensuring concurrency safety.
* **Identity and lineage**, establishing uniqueness and traceability.
* **Metadata and observability**, aligning runtime events with structured monitoring.
* **Errors as events**, preserving analyzability even in failure.
* **Declarative–imperative harmony**, unifying static specifications with dynamic execution.

These axes together define how POP events behave, how they interact with the pipeline graph, and how they remain analyzable in both compile-time and runtime contexts.

#### Immutability

Immutability is a foundational semantic of events in POP. Once an Event<T> object is created, its payload and metadata cannot be altered. This property serves several critical purposes.

First, immutability ensures **concurrency safety**. Because multiple workers may consume and process the same event concurrently, immutability guarantees that no worker can modify state that might affect another. This avoids race conditions and eliminates the need for explicit synchronization, which has long been recognized as a source of nondeterminism and complexity in concurrent systems (Lee & Parks, Dataflow process networks, 1995; Agha, 1986).

Second, immutability facilitates traceability and debugging. Since an event’s identity and lineage metadata are preserved intact, the system can guarantee that recorded traces correspond precisely to the data observed by each node. This property aligns with established best practices in dataflow programming, where tokens are treated as immutable carriers of information, allowing analyzability and reproducibility of program behavior (Dennis & Misunas, 1975).

Third, immutability enhances **optimizations and reasoning**. Compilers and runtimes can safely cache, replicate, or reorder immutable events without risking semantic divergence, as immutability ensures referential transparency. This mirrors principles from functional programming, where immutability simplifies reasoning about program state and correctness (Scott, 2016).

In POP, nodes that need to transform or aggregate information do so by creating *new* events derived from existing ones. Node-local state may exist for caching or batching, but such state never leaks into the event stream. By separating ephemeral worker state from immutable event flows, POP combines efficiency with analyzability.

#### Identity and Lineage

Figure 6: Node Identity and Metadata Lineage

A diagram of events

AI-generated content may be incorrect.

Identity is central to the semantics of POP events. Each Event<T> carries an EventId that uniquely distinguishes it for the duration of a program execution. The EventId is constructed as a tuple of three components:

* The Ingress() node’s identity, assigned by the compiler and stable across builds.
* The worker number within the ingress fleet that processed the event.
* A 64-bit unsigned sequential integer incremented per event per worker during runtime.

This scheme guarantees that events are uniquely identifiable within a single program execution, while remaining deterministic across builds. Sequential numbers reset on each execution, and multiple deployments of the same build share Ingress() node identities. As a result, an EventId is not globally unique, but it will be sufficient for runtime traceability, debugging, and causal analysis or runtime optimization.

Beyond raw identity, POP events also encode lineage. Each Event<T> object maintains references to its causal predecessors, creating an analyzable chain of derivation through the pipeline. This lineage metadata allows developers and tools to trace back the origin of any transformed or aggregated event to the ingress events that produced it. Lineage is vital for debugging, replay, and auditing, and is consistent with established practices in dataflow and distributed tracing systems (Dean & Ghemawat, MapReduce: Simplified data processing on large clusters, 2008; Bjork, Burns, Fong-Jones, & Hochstein, 2020).

The combination of scoped EventIds and lineage metadata ensures that even in the presence of multiple entrypoints, concurrent workers, and complex branching, POP pipelines remain analyzable. Every Event<T> is identifiable, every transformation traceable, and every failure context recoverable.

#### Metadata and Observability

In POP, metadata is a first-class property of every Event<T> object. Alongside its payload, each event carries structured metadata including timestamps, source context, causal history, and execution hints. This metadata plays a role analogous to structured logs, metrics, and traces in distributed systems engineering, forming the foundation for observability.

**Structured Monitoring.** Observability in distributed systems requires multiple complementary data types—metrics for trends, logs for discrete events, and traces for request lifecycles (Beyer, Jones, Petoff, & Murphy, 2016). POP integrates these concerns directly into its event semantics: timestamps and causal context provide traceability, lineage metadata supports distributed tracing, and execution hints allow pipeline-level metrics aggregation by the programmer if desired.

**Noise versus Signal.** Instrumentation is only effective if it balances completeness with clarity. Krabbe (2019) emphasizes that useful observability comes from recording high-signal contextual information without overwhelming operators with redundant data. POP enforces this principle by standardizing per-event metadata: every event carries exactly the information required for debugging, validation, and optimization, no more and no less. Information can be captured offline using FanOut() nodes to split event streams off to a logging Egress() node.

**Detecting Divergence.** Observability also entails catching deviations between specification and execution. Bjork, et al., (2020) note that monitoring must surface both expected success paths and unexpected behaviors. POP’s error-handling model, where Error<E> encapsulates failure details, ensures that divergences are represented as analyzable events within the same semantic framework, not as external anomalies.

By embedding observability into the metadata of every event, POP eliminates the gap between declarative specification and runtime analysis. Developers do not need to add ad hoc logging or external tracing frameworks: every pipeline is instrumented by construction. Metadata thereby becomes a semantic contract that guarantees visibility into event flow, performance, and anomalies across all stages of execution.

#### Errors as Events

Errors are not side channels but first-class events. An Error<E> encapsulates the original event along with error-specific metadata (failure cause, stack trace, diagnostics). This ensures that debugging information remains analyzable, traceable, and tied directly to the original event flow. Error semantics are crucial for POP’s guarantee that failures do not escape the declarative model but are instead observable and routable through error-handling pipelines.

#### Declarative–Imperative Harmony

POP achieves a balance between declarative specification and imperative execution. The declarative layer defines the static pipeline graph: nodes, edges, and dataflow rules. This graph is analyzable at compile time, ensuring type safety, coverage of FanOut() conditions, and absence of forbidden cycles.

At the same time, certain nodes (e.g., Ingress(), Egress(), Transform()) embed imperative worker functions, expressed in LLVM. These functions perform localized computation such as parsing, transformation, or aggregation. To prevent imperative code from undermining declarative guarantees, worker functions are sandboxed:

* They execute in a restricted virtual machine.
* They have no access to global mutable state.
* I/O is only permitted in Ingress() and Egress() nodes, under capability constraints.
* State, where needed for batching or caching, is confined to the scope of a single node and never leaks into the event flow.

This discipline ensures that imperative flexibility remains bounded within a declarative framework. The compiler can reason about pipelines globally, while imperative workers handle localized computation. The result is a division of labor: declarative graphs provide analyzability and safety, while imperative code supplies flexibility where static specifications alone are insufficient.

This integration reflects long-standing discussions in programming language design. Declarative models offer analyzability but limited control, while imperative models provide control at the expense of reasoning complexity (Scott, 2016). POP’s approach resembles hybrid paradigms such as actor systems, where declarative message-passing is combined with imperative actors (Agha, 1986). By carefully constraining imperative behavior, POP reconciles these traditions, ensuring that pipelines remain analyzable, safe, and expressive.

### Event Lifecycle

While event semantics (1.1.6) define what an Event<T> *is*, lifecycle semantics define what an event *does* as it moves through a POP pipeline. From creation at ingress to transformation, routing, error encapsulation, and eventual egress, every event undergoes well-defined state transitions. Explicit lifecycle rules ensure analyzability, reproducibility, and operational clarity, aligning POP with decades of work in dataflow and distributed systems (Dennis & Misunas, 1975; Lee & Parks, Dataflow process networks, 1995; Lamport, 1978).

#### **State Transitions**

In POP, events progress through a finite set of well-defined lifecycle states. Explicit state modeling ensures that event behavior remains analyzable, debuggable, and reproducible across distributed executions. Each Event<T> transitions deterministically from creation to delivery, with failure represented as a structured error event rather than an external anomaly.

The lifecycle of an event can be summarized in the following states:

1. **Created** – An event is instantiated at an Ingress() node, assigned an EventId (comprised of ingress identity, worker number, and a sequential counter), and enriched with metadata such as timestamps and causal context.
2. **Transformed** – Within Transform() nodes, worker functions derive new events from existing ones. Original events remain immutable, preserving concurrency safety.
3. **Routed** – At FanOut() and Collect() nodes, events are directed along declaratively defined edges. FanOut() nodes branch execution paths, while Collect() nodes synchronize streams.
4. **Errored** – If validation or worker execution fails, the original event is encapsulated in an Error<Event<T>>, carrying both the payload and diagnostic metadata. Errors flow through a dedicated error-handling pipeline, ensuring visibility and analyzability.
5. **Delivered** – Events reach an Egress() node, where they are serialized to an external system or persisted, marking the termination of their lifecycle.

This lifecycle may be represented as a finite-state machine:

Figure 7: Event Lifecycle

A diagram of a medical procedure

AI-generated content may be incorrect.

Such modeling reflects long-standing principles in dataflow research. Dennis and Misunas (1975) emphasized token lifecycles in early dataflow processor architectures, while Lee and Parks (1995) formalized process networks as deterministic transitions of tokens between processes. By extending these ideas with structured error encapsulation, POP ensures that every possible transition—success or failure—remains analyzable within the same semantic framework.

The explicit state transition model also strengthens runtime observability. Because each transition updates event metadata, operators can trace the complete journey of an event, correlate failures with their causes, and deterministically replay sequences for debugging. In this way, lifecycle semantics not only enforce correctness but also integrate seamlessly with POP’s guarantees of traceability and reproducibility.

#### Event Ordering

Ordering semantics are a central concern in any event-driven system. Without clear rules, concurrency and distribution can lead to nondeterministic execution, undermining both correctness and analyzability. POP addresses this by defining ordering at the level of ingress, routing, and delivery.

**Per-Ingress Ordering.** POP guarantees sequential ordering of events produced by a given Ingress() node. Each event is assigned a monotonically increasing identifier, ensuring that consumers can reason about event order relative to a specific source. This ordering guarantee is scoped by worker assignment and the EventId sequence, preserving determinism within a single ingress stream.

**Concurrency Effects.** Once dispatched to worker pools, strict global ordering is deliberately relaxed. Multiple events from different ingresses may interleave arbitrarily, and even within the same ingress, parallel workers may reorder events downstream. This relaxation follows the principle that enforcing strict global order in distributed systems incurs prohibitive performance costs (Lamport, 1978).

**Node-Specific Behavior.** Ordering semantics also depend on node type:

* FanOut() nodes preserve per-branch ordering relative to their input, routing each event deterministically along one branch.
* Collect() nodes may reorder events as they merge streams, depending on configuration and runtime scheduling.

**Trade-Offs.** The choice to scope ordering at ingress rather than globally reflects a balance between determinism and throughput. Strict global ordering improves analyzability but limits scalability; relaxed ordering increases concurrency and efficiency but requires developers to design pipelines that tolerate interleaving. This mirrors design choices in distributed stream systems such as Kafka and Flink, which restrict ordering guarantees to partitions or sources rather than entire clusters (Kreps, Kafka: a distributed messaging system for log processing, 2014; Carbone, et al., 2015).

In sum, POP’s ordering model ensures predictability where it matters most—within ingress-defined streams—while allowing flexibility downstream. This strikes a balance between analyzability, concurrency, and performance in distributed event processing.

#### Delivery Semantics

In POP, delivery semantics are tightly coupled to the system’s **backpressure policy** and error handling, which governs how the pipeline responds when downstream nodes cannot process events as quickly as they arrive or when failures occur. This design choice ensures that delivery guarantees are analyzable and predictable at compile time, rather than emergent from ad hoc runtime behavior.

***At-least-once Delivery*.** When a blockpolicy is in place, upstream nodes pause event emission until downstream capacity becomes available. Under this regime, retries are possible, and at-least-once delivery is the strongest guarantee that can be provided without custom worker function logic. Events may be delivered more than once in cases of retry, but none are lost.

***At-most-once Delivery.*** When non-blocking policies such as dropOldest or dropNewest are applied, the system may discard events under overload conditions. In these cases, at-most-once delivery is the only achievable guarantee: an event is either delivered successfully or dropped, but it will never be retried.

**Exactly-once Delivery.** Exactly-once delivery is not a guarantee in declarative POP. It can be achieved through custom logic in worker functions. But as established in distributed systems literature, exactly-once semantics are costly and often infeasible at scale (Skeen, 1983; Kreps, 2014). POP prioritizes analyzability and concurrency safety, opting instead for delivery semantics that are explicit, enforceable, and tied directly to backpressure policy.

**Error Handling.** When a node cannot deliver or process an event within defined constraints, the event is wrapped in an Error<Event<T>> and routed to the error-handling pipeline. This ensures delivery violations are visible, analyzable, and recoverable, rather than silently discarded.

By binding delivery semantics to backpressure, POP integrates **performance, safety, and correctness** into a unified design axis. Developers can reason about guarantees statically, knowing that the backpressure choice simultaneously declares the delivery model. This mirrors practices in modern streaming frameworks, where at-least-once semantics coupled with structured error pipelines provide robustness without the overhead of global exactly-once enforcement (Carbone et al., 2015).

#### Event Reproducibility and Replay

Immutability and lineage metadata give POP the theoretical capability to reproduce and replay event flows. In principle, replay can serve useful purposes such as debugging, simulation, and validation. By reintroducing events with preserved EventId and lineage information, developers could reconstruct past executions and analyze pipeline behavior under controlled conditions.

However, replayability introduces significant security risks. Allowing arbitrary re-injection of events opens the door to replay attacks, duplication of sensitive operations, or unauthorized reprocessing of confidential data (Syverson, 1994; Kaufman, Perlman, & Speciner, 2002). For these reasons, the POP standard discourages reproducibility and replayability in production contexts.

Instead, POP treats reproducibility and replay as an *optional debugging mode* that may be enabled in controlled environments. When enabled, this mode must be explicitly declared, constrained to non-production pipelines, and subject to capability restrictions. In such cases, replay events should be accompanied with conspicuous warnings to ensure the user is aware of this insecure status in observability and auditing systems.

By separating replay as a non-standard, insecure feature, POP provides a controlled path for testing and analysis while maintaining strong security and integrity guarantees in normal operation.

### Error Handling Pipelines

POP provides structured error handling via special error handling pipelines, which are declared at compile time and processed like any other POP pipeline. This mechanism ensures that errors are not accidental or ad hoc, but predictable, analyzable, and observable.

#### Default Error Pipeline Behavior

Every POP program includes a built-in error handling pipeline of last resort, formally designated as ErrorPipeline(). This pipeline is responsible for processing all unhandled error events, defined as Error<Event<T>> (or Error<E> for short), where the wrapper encapsulates the original event (E) alongside error-specific metadata. By design, the existence of a default error pipeline ensures that no error is ever silent or invisible: all failures are routed through a predictable, analyzable path.

The default ErrorPipeline() is configured with a terminal Egress() node that writes error information to stderr and terminates program execution upon receipt of an unhandled error. This behavior reflects the principle of fail-fast design, where unrecoverable errors are surfaced immediately and program state is not left in a potentially inconsistent condition (Miller, 1986; Saltzer & Kaashoek, Principles of Computer System Design: An Introduction., 2009).

Although ErrorPipeline() is predefined, it remains a **compile-time structure** subject to the same guarantees as any other pipeline: type safety, immutability, backpressure, and observability. Because the error pipeline is declarative, the compiler can analyze it, enforce structural constraints, and ensure that all possible error flows are explicitly accounted for. This mirrors structured error handling in programming languages, where exceptions or faults are captured within statically analyzable constructs rather than left as unchecked runtime anomalies (Cardelli, Type Systems, 1989; Leroy, 1992).

Importantly, developers are not bound to the default stderr behavior. The declarative model permits redefinition of the Egress() node attached to ErrorPipeline(), allowing error events to be written to persistent storage, transmitted across networks, or processed in more complex ways. In this sense, ErrorPipeline() is both a safeguard of last resort and a flexible extension point for program-specific error handling strategies.

#### Custom ErrorPipeline Definitions

While ErrorPipeline() provides a safe default, POP also allows developers to **extend and customize error pipelines** declaratively. This enables programs to define more sophisticated handling strategies, including retries, classification, redirection, or logging to alternate destinations.

A key feature of error pipeline customization is the ability to **classify errors**. POP introduces a set of built-in runtime error categories, which serve as the foundation for error analysis and routing:

Table 1: Runtime Error Classes

|  |  |
| --- | --- |
| ParseError | Event cannot be parsed into the expected schema or type. For example, malformed JSON in an input stream. |
| ValidationError | An event is well-formed syntactically but fails semantic validation, such as failing a type constraint or exceeding a numeric limit. |
| IOError | Input/output operations fail, such as file system errors, broken network connections, or failed writes to external sinks. |
| BackpressureEvent | An edge experiences backpressure, indicating that downstream nodes cannot keep up with event throughput. Metadata captures start and stop times of the condition for analysis. |
| WorkerRuntimeError | A worker function fails with an unhandled runtime error (e.g., segmentation fault, arithmetic exception). |
| WorkerCapacityReached | Raised when a worker pool has reached maximum capacity and new workers cannot be added. |
| WorkerLaunchFailed | Raised when a worker’s virtual machine (or sandbox) cannot be launched, typically due to resource exhaustion or configuration errors. |
| CustomError | A program-defined classification for domain-specific errors not covered by built-in categories. Developers may extend this type to capture application-level semantics (e.g., BusinessRuleViolation). |

When Error<E> objects are classified, they can be routed within the ErrorPipeline() to specialized handlers. For instance, transient errors such as IOError or BackpressureEvent may be retried with exponential backoff, while irrecoverable errors such as ParseError may be redirected immediately to a dead-letter sink. This mirrors real-world best practices in fault-tolerant dataflow systems, which emphasize classifying recoverable versus non-recoverable errors for operational resilience (Carbone, et al., 2015; Digibee, 2023).

Because error pipelines are compile-time constructs, classification logic is statically analyzable. The compiler ensures that all branches of the error pipeline are well-typed, that no error classes are silently dropped unless explicitly configured, and that observability hooks remain intact. This reflects broader programming language research emphasizing structured, type-safe exception handling (Leroy, 1992; Miller, 1986).

#### Interaction with Retries and Backpressure

Error handling pipelines in POP do not exist in isolation; their behavior is intertwined with the system’s backpressure policy. Because delivery semantics (see §1.1.7.3) are derived from the backpressure model, retry strategies within error pipelines must be consistent with the guarantees offered by that model.

**Blocking policies.** When the pipeline employs a block policy, upstream nodes pause emission until downstream capacity is available. In this regime, error pipelines may implement retrysemantics for transient errors such as IOError or BackpressureEvent. For example, an Error<E> caused by a failed network write can be re-submitted after a backoff period. This aligns with at-least-once delivery: the same event may be delivered more than once, but no event is silently lost.

**Non-blocking policies.** In contrast, under non-blocking policies such as dropNewest, dropOldest, or shunt, retries may be ineffective. An event that is retried could again be discarded immediately if capacity constraints persist. In these cases, error pipelines can only guarantee at-most-once delivery, and must rely on explicit dead-letter handling to preserve visibility. Thus, POP requires developers to design error pipelines that distinguish between transient errors worth retrying and conditions where retrying will only compound failure.

**Retry orchestration.** Because error pipelines are declarative and compile-time analyzable, retry behavior can be explicitly encoded in the pipeline structure. For example, a retry loop might be represented as a bounded sub-pipeline with exponential backoff, capped attempt counts, and fallback routing to a dead-letter sink. This structured retry handling prevents the common pitfalls of ad hoc retry loops in imperative systems, such as infinite retries or retry storms (Carbone, et al., 2015).

**Observability of retries.** Each retry attempt generates an Error<E> event with updated metadata, including retry count, elapsed time, and cause classification. This ensures that retry behavior is not opaque to operators: it can be traced, monitored, and audited as part of the normal observability pipeline. Such visibility reflects best practices in fault-tolerant distributed systems, where retries must be both bounded and transparent (Kleppmann, 2017).

By linking error pipeline behavior to backpressure, POP ensures that retry semantics remain analyzable, safe, and consistent with the overall delivery model. This design prevents mismatches where error handling might promise guarantees that the underlying backpressure regime cannot provide.

#### Observability and Error Semantics

In POP, error handling pipelines are not opaque recovery mechanisms but first-classobservabilityconstructs. Every error is represented as an Error<E>, which encapsulates the original event along with diagnostic metadata such as error type, timestamp, retry count, and causal lineage. This ensures that the context of the failure is never lost: operators and tools can always trace the path of an error back to its source and intermediate transformations.

**EventId and Lineage.** Because Error<E> preserves the original event’s EventId and lineage metadata, errors remain embedded within the same causal graph as normal events. This enables deterministic tracing, where operators can reconstruct not only where the error occurred but also which ingress produced the event and how it traversed the pipeline. This aligns with established distributed tracing practices, where identifiers and causal chains are the foundation of observability (Beyer, Jones, Petoff, & Murphy, 2016; Bjork, Burns, Fong-Jones, & Hochstein, 2020)

**Structured Error Metadata.** Error classification (see §1.1.8.2) enriches observability by distinguishing transient errors (e.g., IOError, BackpressureEvent) from terminal errors (e.g., ParseError, ValidationError). Each classification yields structured metadata, making errors analyzable as datasets rather than unstructured logs. This practice mirrors approaches in modern observability systems, which emphasize high-signal structured data over raw logging (Krabbe, 2019).

**Operational Visibility.** By default, all unhandled errors flow into ErrorPipeline(), which exposes them via an Egress() node. In custom configurations, this may mean persistence in error logs, forwarding to monitoring infrastructure, or even integration with alerting systems. Crucially, POP’s declarative model ensures that such observability guarantees are built into the language, not bolted on as an afterthought. This approach reflects the principle articulated in Google’s SRE practice: reliability requires observability as a first-class design goal, not as an operational patch (Beyer, Jones, Petoff, & Murphy, 2016).

**Semantic Uniformity.** Treating errors as analyzable events prevents the fragmentation of observability. Rather than relying on ad hoc exceptions, logging frameworks, or external monitoring, errors in POP follow the same semantic pathways as normal events. This uniformity simplifies debugging, enhances reproducibility, and ensures that every error is visible and traceable across the system.

#### Trade-Offs and Best Practices

Error handling pipelines in POP provide a structured, analyzable way to manage failures, but they also introduce trade-offs that must be carefully considered.

**Performance Overhead.** Introducing retries, classification, or complex routing in error pipelines can increase system latency and resource consumption. In high-throughput systems, retry storms or excessive error logging can themselves cause backpressure or secondary failures (Kleppmann, 2017). Developers should apply bounded retries with exponential backoff and enforce limits on error-processing overhead.

**Complexity versus Clarity.** While POP encourages rich error-handling strategies, overly complex error pipelines may obscure system behavior. Research in programming languages has shown that unstructured exception handling can reduce program comprehensibility and analyzability (Miller, 1986). POP mitigates this risk by enforcing declarative structure, but best practice is to maintain simplicity: classify errors into a manageable set of categories and ensure explicit routing.

**Risk of Silent Failure.** Improper configuration of error pipelines can suppress or discard errors without sufficient visibility, undermining POP’s observability guarantees. As Saltzer and Kaashoek (2009) emphasize in their discussion of system design principles, errors must be made visible to operators to prevent hidden state inconsistencies. Dead-letter sinks or persistent error logs should be used as fail-safe mechanisms to avoid silent loss of diagnostic information.

**Alignment with Delivery Semantics.** Error-handling strategies must be consistent with the delivery guarantees imposed by backpressure policies (see §1.1.7.3). For example, retries only have meaning under blocking policies that allow at-least-once delivery, whereas under drop policies, retries are ineffective. Best practice dictates explicit documentation of the chosen backpressure policy and its implications for error handling.

**Security Considerations.** Error pipelines may expose sensitive payloads, stack traces, or operational metadata. As Kaufman, Perlman, and Speciner (2002) note in their treatment of system security, uncontrolled error reporting can leak information exploitable by adversaries. POP programmers should ensure that error pipelines sanitize or redact sensitive fields before forwarding them to external sinks.

In sum, effective use of error pipelines requires balancing robustness with simplicity, ensuring that retries and classification strategies are bounded, observable, and consistent with the underlying delivery model. By adhering to these practices, developers can leverage error pipelines to improve reliability without compromising performance, security, or clarity.

## Event Source Strategies

This section introduces how POP pipelines admit data into the system. Having defined event semantics (1.1.6), lifecycle (1.1.7), and error handling (1.1.8), the next natural step is to address where events originate, how Ingress() nodes interact with external systems, and what security/trust boundaries are enforced.

### **Ingress Sources**

In POP Ingress() nodes define the entrypoints through which external data becomes events within a pipeline. Because Ingress() is the only node type permitted to perform input operations from external sources, it establishes both a functional and security boundary between the external environment and the declarative pipeline graph.

Ingress()sources can be broadly categorized into three main types:

* **Subscriber-based** — The Ingress() node subscribes to an operating system or middleware event hook. When a triggering event occurs, the system notifies the node, which launches a worker function to capture the event data.
* **Timer-based** — An Ingress() node subscribes to a timer, executing its worker function at a specific time or interval. Timer-based ingress is suited to periodic tasks such as scheduled file imports or monitoring checks.
* **Polling** —The Ingress() node repeatedly executes its worker function to perform I/O operations against a resource. Polling ingress is appropriate for systems without event hooks, though it incurs higher overhead compared to subscriber or timer models.

Ingress()nodes may consume information from files, devices, network services, or other resources through the operating system API. Worker functions perform these data acquisition operations, but always within the limits of a declaredcapabilityset that constrains what kinds of I/O can be performed. This declaration ensures analyzability at compile time and prevents unsafe or unbounded I/O behavior.

The Ingress()node must be present for a POP program pipeline as well as for reusable modules defined as distributable packages. In this case, the pipeline segment can be tested with a test Ingress()node then implemented in a final program using a subscriber Ingress()or Collect() node. In fact, the test harness signal generator could be a simple timer-based Ingress()node used to generate input signal data during pipeline segment development.

Thus, ingress nodes in POP serve as formalized, analyzable, and secure bridges between external systems and declarative pipelines. By constraining ingress behavior to subscriber-based, timer-based, or polling models—and ensuring that modules remain testable without special ingress types—POP maintains both clarity and rigor in defining event sources. This is consistent with established literature (Hohpe & Woolf, 2004).

### **Source Capabilities and Constraints**

Ingress() nodes in POP not only define how events are acquired but also declare **capabilities** and **constraints** that govern their interactions with external resources. This model separates compile-time guarantees from runtime enforcement, ensuring analyzability, predictability, and safety.

#### Compile-Time Capability Declarations.

At compile time, every Ingress() node must explicitly declare the categories of I/O operations it is permitted to perform. These are drawn from a fixed vocabulary of operation classes, such as:

* FileRead – permission to read file contents.
* FileList – permission to enumerate individual files.
* DirectoryList – permission to enumerate directory contents.
* NetworkIn – permission to accept inbound network connections.
* DeviceRead – permission to acquire input from devices such as sensors.

This vocabulary ensures that ingress behavior is analyzable in advance, similar to the role of type systems in bounding program behavior (Cardelli, Type Systems, 1989). The compiler enforces these declarations strictly: a pipeline with no declared FileRead capability cannot compile an ingress worker that attempts to read files.

#### Runtime Modifiers for Access Control.

While capabilities establish coarse-grained categories, ingress nodes may further refine permissions through runtimemodifiers that constrain the scope of allowed operations.

Examples include:

* FileRead:/opt/\*.json — restrict file read operations to JSON files in a specific directory.
* NetIn:127.0.0.1:8080 — restrict inbound network events to localhost on port 8080.
* NetIn:0.0.0.0:22 — permit ingress from all interfaces but only on port 22.

These modifiers act as declarative runtime policies: the pipeline runtime enforces them automatically, preventing ingress workers from exceeding their declared permissions. This aligns with the principle of least privilege and capability-based security models (Hardy, 1988; Levy, 1984).

#### Constraints and Safety.

Ingress declarations also specify performanceandsafetyconstraints such as throughput limits, retry budgets, and timeout policies. For example, an API ingress may be limited to 100 requests per second, while a device ingress may specify a maximum sampling rate. The compiler verifies these constraints and produces errors or warnings if downstream pipeline assumptions are violated.

#### Integration of Compile-Time and Runtime Guarantees.

By combining compile-time capability declarations with runtime modifiers, POP ingress nodes achieve a dual guarantee:

1. **Static analyzability**, ensuring that ingress behavior can be reasoned about formally before execution.
2. **Dynamic enforcement**, ensuring that the runtime prevents unauthorized or unsafe operations.

This layered approach mirrors best practices in secure system design, where static verification is paired with runtime enforcement to close gaps between specification and implementation (Kaufman, Perlman, & Speciner, 2002).

### **Source Isolation and Sandboxing**

#### Worker Function Virtual Machines

A defining property of Pipeline-Oriented Programming (POP) is that all node worker functions execute inside **sandboxed virtual machines**. This design choice enforces strict separation between the declarative pipeline graph and the imperative worker code, ensuring analyzability, concurrency safety, and security.

#### Compiler-Optimized Virtual Machines.

At compile time, the POP compiler analyzes worker functions and their declared capabilities to synthesize a **minimal virtual machine (VM)** for each node. Only the features required by the worker function, combined with the capability set explicitly granted to the ingress node, are included. For example:

* A worker function with FileRead:/opt/\*.json capability will compile into a VM that supports file reading only within the defined scope.
* A worker with NetIn:127.0.0.1:8080 capability will include network sockets but restricted to localhost port 8080.
* Functions without I/O capabilities will be compiled into minimal VMs stripped of all system call interfaces.

This “least features” approach mirrors the principle of least privilege in operating system security (Saltzer & Schroeder, The protection of information in computer systems, 1975), but enforced at the language and compiler level.

#### Isolation at Runtime.

During execution, each VM instance is isolated from both the host system and other VMs. Isolation mechanisms ensure that:

* Worker functions cannot escalate privileges beyond declared capabilities.
* Memory and state are confined within the VM boundary.
* Worker function code integrity is guaranteed. Code and data are consistently separate security contexts.
* Communication between nodes occurs only through typed events, not side channels or shared global state.

This architecture parallels capability-based operating system designs (Levy, 1984) and modern containerization approaches, where a combination of isolation and minimal privilege reduces the attack surface.

#### Determinism and Observability.

Sandboxing also reinforces determinism. Because VMs lack undeclared system interfaces, the behavior of worker functions is constrained and predictable. Moreover, observability hooks are inserted at the VM boundary, ensuring that all I/O and state transitions can be traced as part of the pipeline’s metadata layer (Mace, Roelke, & Fonseca, 2015).

By combining compiler-optimized minimalism with runtime sandboxing, POP ensures that source ingress and worker functions remain both safe and analyzable. This dual enforcement mechanism prevents accidental misuse, supports rigorous reasoning about system behavior, and creates a hardened execution environment for event pipelines.

### **Security and Trust Boundaries**

Security in Pipeline-Oriented Programming (POP) arises from the explicit separation of declarative pipelines from **imperative** worker functions and the strict enforcement of declared capabilities. However, security cannot be fully prescribed at the paradigm level; some concerns are handled by the language runtime, while others depend on the programmer’s operational and domain-specific choices. To clarify these dimensions, POP distinguishes between language-enforced trust boundaries and implementation-specific policies.

#### **Language-Enforced Boundaries**

POP defines strict language-level boundaries to reduce the attack surface of programs:

* **Ingress-only I/O.**

All external inputs enter exclusively through Ingress() nodes, while all outputs leave through Egress() nodes. This ensures that I/O interactions remain analyzable and explicitly declared, eliminating hidden or ad hoc I/O channels.

* **Capability enforcement.**

Capabilities (e.g., FileRead, NetIn) and runtime modifiers (e.g., FileRead:/opt/\*.json) are declared at compile time and enforced at runtime. This aligns with capability-based security principles, where authority is tied to explicit rights over objects (Levy, 1984).

* **Sandbox isolation.**

Worker functions execute inside compiler-optimized virtual machines that contain only the minimal features needed for declared operations. This prevents privilege escalation and follows the principle of least privilege (Saltzer & Schroeder, The protection of information in computer systems, 1975).

#### **Trust Zones and External Sources**

POP requires that trust boundaries be modeled explicitly in pipeline graphs. For example, an Ingress() node consuming telemetry from localhost may be marked trusted, while one listening on a public socket (e.g., NetIn:0.0.0.0:22) must be treated as untrusted. Pipelines cannot silently merge trusted and untrusted inputs; any such interaction requires explicit Transform() validation. This explicit modeling makes trust boundaries visible and analyzable, a departure from traditional event-driven architectures where trust assumptions are often implicit (Schneider, 2000).

#### **Policy-Driven Security Decisions**

Not all security requirements can be embedded directly into the language. POP intentionally leaves some aspects to policy decisions:

* **Authentication and encryption.**

POP permits worker functions to implement cryptographic protocols, but it does not prescribe specific algorithms, which remain domain- and context-dependent.

* **Auditing and logging.**

Metadata and error pipelines support observability, but the scope of audit logging and retention policies are determined by operational needs.

* **Data retention and privacy.**

POP enforces immutability of events but leaves broader compliance and minimization strategies to the programmer and system operator.

This separation reflects a deliberate balance: POP enforces universal invariants while allowing flexibility for domain-specific policies.

#### **Attack Surfaces and Mitigations**

POP reduces attack surfaces by design but recognizes several threat vectors:

* **Replay attacks.**

By default, POP discourages reproducibility and replayability of events (see Section 1.1.7.4). Enabling replayability creates an “insecure” debugging mode unsuitable for production.

* **Ingress exploitation.**

Attack surfaces at ingress points are narrowed through capabilities and runtime modifiers. For instance, NetIn:127.0.0.1:8080 is safer than NetIn:0.0.0.0:\*.

* **Worker compromise.**

Even if a worker is compromised, sandboxed VMs restrict its authority to declared capabilities. Moreover, workers are stateless, further reducing long-term risk exposure.

#### **Compositional Security**

Security in POP is compositional: the secure properties of individual pipelines extend across package boundaries when multiple pipelines are combined. Each node retains its declared trust level and capability set, preventing emergent vulnerabilities that often arise when modules interact without global coordination (Schneider, 2000). This ensures that security guarantees are preserved not only locally but system-wide.

### **Design Trade-Offs**

Like any paradigm, Pipeline-Oriented Programming (POP) embodies a set of deliberate trade-offs. These choices balance competing priorities such as security, performance, and flexibility. While POP enforces strict invariants (e.g., immutability, ingress-only I/O, sandboxed execution), many practical consequences emerge when these principles intersect with real-world requirements for integration, latency, and scalability.

#### **Richer Source Integrations and Risk Surface**

POP’s support for diverse ingress sources—including subscriber-based, timer-based, and polling mechanisms—enables integration with a wide array of systems. However, broader integration increases the potential attack surface. Each additional source type introduces new vectors for exploitation or failure, ranging from malformed data ingestion to timing-based attacks. The paradigm mitigates this risk by confining all external I/O to Ingress() nodes and enforcing declared capability sets (Levy, 1984). Nevertheless, the decision to incorporate richer sources must be balanced against the corresponding expansion in risk surface, particularly in security-sensitive domains.

#### **Stricter Validation and Performance Costs**

POP enforces strong validation at pipeline boundaries, with all events and errors (Event<T>, Error<E>) carrying metadata that supports type checking, backpressure policies, and observability. Stricter validation yields stronger safety guarantees, ensuring pipelines fail visibly and predictably rather than silently propagating errors. Yet, this rigor introduces latency: each validation step consumes computational resources, and aggressive schema checks may delay event processing in high-throughput systems (Saltzer & Schroeder, The protection of information in computer systems, 1975). The trade-off reflects the classic tension between safety and performance in distributed programming (Schneider, 2000).

#### **Balancing Flexibility and Analyzability**

POP aims to balance these opposing pressures by allowing broad source diversity while maintaining analyzability and concurrency safety through language-enforced constraints. Type safety, sandbox isolation, and explicit trust boundaries ensure that pipelines remain predictable, even as they integrate heterogeneous sources. This balance reflects a core principle of POP: expressive power is intentionally constrained to achieve stronger guarantees in areas that are traditionally weak in event-driven systems, including concurrency determinism, fault isolation, and security.

## Pipeline Composition

While individual pipelines in Pipeline-Oriented Programming (POP) are designed as analyzable and secure units, real-world applications frequently require their integration into larger assemblies. In contrast to ad hoc event-driven systems, where composition is often an emergent implementation detail, POP treats pipeline composition as a **first-class semantic concept**. This ensures that correctness, security, and analyzability extend across pipeline boundaries, even when pipelines span multiple packages or trust domains.

### **Composition by Package Integration**

Pipelines in POP can be defined independently within package namespaces and later composed at compile time. This is accomplished by using Collect() nodes as entry points for downstream segments, enabling one pipeline to feed into another while maintaining separation of concerns. Each pipeline segment preserves its declared namespace, capability set, and trust zone, preventing unintentional privilege escalation or type mismatches.

The compiler ensures type-safe composition by instantiating pipeline segments independently when required, preserving distinct event typing across segments (see Section 1.1.4.2). This prevents semantic drift that often arises in loosely typed or dynamically wired event systems.

### **Event Typing and Compatibility**

Event typing provides the foundation for pipeline interoperability. For composition to succeed, the output event type of one segment must align with the input type of the next. POP enforces these contracts at compile time, rejecting mismatches unless explicitly bridged through a Transform() node.

This approach parallels type-safe function composition in functional programming, where type mismatches are surfaced during compilation rather than deferred to runtime (Scott, 2016). By elevating type compatibility to a compile-time requirement, POP avoids the class of runtime failures common in message-passing systems where event schemas are implicitly assumed.

### **Error Pipeline Integration**

Error handling pipelines (ErrorPipeline()) are also compositional. Each package defines its own error-handling path, ensuring that local failures are addressed near their origin. However, composed pipelines can escalate unhandled Error<E> objects into a global supervisory error pipeline.

This hierarchical error model prevents errors from being silently discarded while also enabling separation of concerns: local packages handle routine errors internally, while systemic or security-critical errors are routed to global handlers. Such explicit integration of error handling contrasts with conventional systems, where error flows often lack compositional semantics and must be manually orchestrated (Carbone, et al., 2015).

### **Trust and Capability Propagation**

When pipelines span trust boundaries, POP enforces explicit validation at composition points. Ingress and transform nodes retain their declared trust levels, and capabilities are not implicitly propagated downstream. Each composed pipeline segment must declare its own capabilities independently, ensuring that a downstream segment cannot inherit or widen the authority of an upstream segment.

This design prevents “capability creep,” where security assumptions degrade silently as systems grow in complexity (Levy, 1984; Schneider, 2000). By requiring explicit declarations, POP maintains analyzability and preserves the principle of least privilege across the composed graph.

### **Package Versioning**

In modular systems, versioning is a persistent source of fragility, particularly when multiple pipeline packages evolve independently. POP addresses this challenge by making versioning explicit at the level of pipeline composition.

**Compile-Time Binding.** Package versions are resolved and bound at compile time, not deferred to runtime. This ensures that pipelines composed from multiple packages use a consistent set of dependencies. If two segments require incompatible versions of the same package, the compiler detects the conflict and prevents ambiguous or unsafe compositions.

**Compatibility Contracts.** Event typing and schema contracts provide a natural mechanism for distinguishing between backward-compatible and breaking changes. Backward-compatible changes, such as extending an event schema with optional fields, allow pipelines to continue functioning without modification. Breaking changes, such as altering type signatures, result in compile-time errors that force explicit reconciliation, often through Transform() nodes.

**Multi-Version Composition.** When different versions of a package must coexist, POP requires explicit disambiguation. Each version is assigned a distinct namespace, and composition across versions is mediated by typed transforms. This prevents accidental reliance on ambiguous or inconsistent package behavior, a problem well documented in traditional module systems (Cardelli, Type Systems, 1989).

**Security and Trust Implications.** Because capabilities and trust levels are tied to specific package declarations, versioning also affects authority propagation. Older versions of a package cannot silently escalate privileges if newer versions restrict them; each version must independently declare its capabilities, preserving analyzability across upgrades.

By embedding version resolution into the compilation process, POP transforms package versioning from a source of runtime fragility into a statically analyzable property. While this imposes stricter constraints on composition, it ensures that pipeline assemblies remain secure, predictable, and formally verifiable.

### **Operational Considerations**

Compositional semantics in POP extend beyond type and security to operational behavior. Worker-pool isolation is preserved across segments, preventing cross-contamination of concurrency domains. Backpressure policies also compose explicitly: when upstream and downstream pipelines declare conflicting policies, the compiler requires explicit resolution rather than defaulting silently.

Observability is supported across pipeline boundaries through propagation of EventId metadata. This ensures that event traces can be followed seamlessly through the composed system, preserving the end-to-end observability that underpins debugging, performance optimization, and compliance auditing (Mace, Roelke, & Fonseca, 2015).

### **Benefits and Limitations of Composition**

Pipeline composition in POP offers strong benefits but also introduces certain limitations. On the benefits side, composition enables **modularity and reuse**: developers can define pipelines within packages, then integrate them into larger systems without sacrificing type safety or analyzability. By making composition a first-class semantic construct, POP avoids the emergent fragility that characterizes many ad hoc event-driven systems. Observability is preserved across composed graphs through EventId propagation, while security boundaries remain analyzable thanks to explicit capability and trust declarations (Levy, 1984; Schneider, 2000).

At the same time, limitations emerge from POP’s strict guarantees. Because type compatibility and capability declarations are enforced at compile time, composition may feel restrictive compared to more flexible—but less safe—systems. Backpressure policies, when misaligned between segments, require explicit resolution by the developer, adding up-front complexity. Similarly, error pipelines must be reconciled across segments, which increases design overhead but ensures that no error paths are lost.

These trade-offs reflect POP’s central philosophy: **constraints are intentionally chosen to yield analyzability, security, and determinism at scale.** While composition in POP demands discipline from developers, it produces pipelines that are both reliable and formally verifiable in ways that traditional event-driven architectures cannot achieve.

## Concurrency, Parallelism and Event Loops

Figure 8: Pipeline Loops

A diagram of a loop

AI-generated content may be incorrect.

### Concurrency and Parallelism Distinguished

Concurrency and parallelism are often conflated, but they represent distinct concepts in computing systems. Concurrency refers to structuring a system so that multiple tasks make progress independently, interleaving execution to deal with many things at once. Parallelism, in contrast, refers to executing multiple tasks simultaneously, typically leveraging multiple cores or processors to do many things at once (Pike, 2012).

This distinction is foundational for Pipeline-Oriented Programming (POP). POP treats concurrency as the default abstraction: events (Event<T>) flow through pipelines independently, with immutability ensuring safe concurrent execution. Parallelism then emerges as a runtime optimization, where worker pools and event loops may process multiple events simultaneously when hardware resources allow. By separating these concerns, POP avoids the pitfalls of shared mutable state found in traditional threading models while enabling analyzable, predictable concurrency.

### Parallelism Presumption

Pipeline-Oriented Programming (POP) presumes parallelism as a baseline property of event execution. Because Event<T> objects are immutable and carry unique EventId identifiers, multiple events can be processed simultaneously across workers without risking data races or inconsistent states. This presumption removes the burden from the programmer of reasoning about low-level thread interleavings or shared mutable state, a common source of errors in traditional concurrent programming models (Lee, The problem with threads, 2006).

The POP runtime exploits hardware parallelism whenever available, dispatching events to worker pools assigned to each node. Workers may execute on different cores, processors or even different machines, but correctness is preserved because pipelines enforce immutability and explicit merge semantics at Collect() nodes. This design achieves what (Pike, 2012) described as the separation of concerns: concurrency is the structural property of the system, while parallelism is the performance-oriented execution strategy. In POP, developers design pipelines as concurrent pipeline graphs, and the runtime parallelizes execution opportunistically.

This presumption of parallelism also has implications for determinism. Unlike thread-based programming, where interleavings can cause nondeterministic behavior, POP constrains nondeterminism through strong guarantees:

1. **Immutability** ensures events cannot be altered once emitted.
2. **Scoped mutability** is only permitted through the mutable() construct, confining state changes to tightly controlled operations.
3. **Explicit ordering** at merge points prevents ambiguity in pipeline outputs.

Thus, POP balances the benefits of parallel execution with analyzability and correctness. While developers may not control the exact scheduling of events across workers, they gain a programming model that scales naturally with hardware parallelism while preserving formal guarantees of safety and determinism.

### Event-Level Concurrency

Concurrency in POP rests on the principle that both events and functions are immutable by default. Each Event<T> object, once emitted, cannot be altered, and no mutable state is shared across nodes, functions, or scopes. Worker functions themselves are side-effect free, ensuring concurrency safety as a property of the paradigm rather than an implementation detail.

When mutability is required, it must be explicitly invoked using the mutable() construct. This confines mutations to a narrow lexical scope and prevents side effects from leaking into the wider pipeline. By restricting mutability to explicit, scoped operations, POP avoids the accidental introduction of shared state—a problem endemic to thread-based concurrency (Lee, 2006).

#### **Immutable Events and Scoped Mutability**

Immutability is the cornerstone of concurrency safety in Pipeline-Oriented Programming (POP). Every Event<T> object is immutable from the moment it is emitted by an Ingress() node until it is consumed at an Egress() node. This property guarantees that concurrent execution of worker functions cannot result in data races or unintended side effects, because no event can be modified once placed into the pipeline. Immutability at the event level is complemented by immutability within worker functions themselves: scopes within functions are immutable by default, and variables cannot be reassigned once bound.

To address the practical need for stateful computations, POP introduces mutability only through the explicit mutable() construct. This construct enables developers to mark a specific operation as mutable, confining state changes to a narrow lexical and temporal scope. For example, a worker may aggregate values or update a local counter within the boundaries of a mutable() block, but these changes cannot propagate beyond the function or alter the immutable event stream.

The design of mutable() is consistent with established practice in programming language research, where absolute purity has often been relaxed through carefully scoped mechanisms. Functional languages, for example, introduce controlled side-effect models such as monads to reconcile purity with practical expressiveness (Peyton Jones, 2003). Similarly, early work in dataflow programming emphasized that while immutability enables analyzability and determinism, bounded mutable state can coexist if its scope is constrained and formally analyzable (Dennis & Misunas, 1975; Lee & Parks, Dataflow process networks, 1995). Pragmatic treatments of language design also highlight that carefully restricted mutability strikes a balance between usability and safety (Scott, 2016).

By enforcing immutability as the default and mutability as an explicit, scoped opt-in, POP avoids the pitfalls of traditional concurrency models. In thread-based systems, shared mutable state requires synchronization mechanisms such as locks, semaphores, or transactional memory, each of which introduces complexity and opportunities for deadlock or race conditions (Lee, The problem with threads, 2006). POP sidesteps these hazards by making immutability a language-level guarantee, ensuring that concurrency safety is built into the paradigm rather than layered on through defensive programming.

In sum, POP’s mutable() feature reflects a well-defended design choice in programming languages: it preserves the analyzability and determinism afforded by immutability while providing the flexibility necessary for practical computations within tightly defined boundaries.

Immutability is the cornerstone of concurrency safety in Pipeline-Oriented Programming (POP). Every Event<T> object is immutable from the moment it is emitted by an Ingress() node until it is consumed at an Egress() node. This property guarantees that concurrent execution of worker functions cannot result in data races or unintended side effects, because no event can be modified once placed into the pipeline. Immutability at the event level is complemented by immutability within worker functions themselves: scopes within functions are immutable by default, and variables cannot be reassigned once bound.

To address the practical need for stateful computations, POP introduces mutability only through the explicit mutable() construct. This construct enables developers to mark a specific operation as mutable, confining state changes to a narrow lexical and temporal scope. For example, a worker may aggregate values or update a local counter within the boundaries of a mutable() block, but these changes cannot propagate beyond the immediate scope, except as a new object copied from the original. This design preserves analyzability and determinism, while allowing limited flexibility where mutable operations are essential.

By enforcing immutability as the default and mutability as an explicit, scoped opt-in, POP avoids the pitfalls of traditional concurrency models. In thread-based systems, shared mutable state requires synchronization mechanisms such as locks, semaphores, or transactional memory, each of which introduces complexity and opportunities for deadlock or race conditions (Lee, 2006). POP sidesteps these hazards by making immutability a language-level guarantee, ensuring that concurrency safety is built into the paradigm rather than layered on through defensive programming.

This approach resonates with the broader trends in functional and dataflow programming, where immutability underpins determinism and analyzability (Lee & Parks, 1995). POP extends these principles with a controlled escape hatch for mutability, striking a balance between rigorous concurrency safety and practical expressiveness.

#### **Node-State Tables**

While POP enforces immutability across events and worker scopes, some computations—such as aggregation, caching, or coordination—require limited shared state. To support these operations without undermining concurrency safety, POP introduces node-state tables.

A node-state table is available only within the scope of a single node, typically a Transform(). It enables workers attached to the same node to coordinate on shared state, while preventing leakage across the broader pipeline. Importantly, the table is never directly exposed; it is accessed only through four methods:

* get(key) – retrieve a value associated with a compile-time–declared key.
* set(key, value) – assign a new/updated value to a key, enforcing type safety.
* update(key, value) – atomically update an existing key’s value.
* list() – enumerate declared keys and their current bindings.

This API-based access model enforces atomicity and ensures that state changes are concurrency-safe. Internally, the runtime applies key-level locking, so only one worker may modify a key at a time. If a lock or operation does not complete within the compiler-defined stateTableTombstone window, the transaction is rolled back and the pipeline emits an Error<E>. This approach draws from transactional memory and database concurrency control research, where atomicity and rollback are central to safety under contention (Gray & Reuter, 1993; Herlihy & Moss, Transactional memory: Architectural support for lock-free data structures., 1993).

Node-state tables are also ephemeral: their contents exist only in memory and are discarded when the program terminates. Keys are declared at compile time and are strongly typed, ensuring analyzability, while values may evolve dynamically at runtime. This reflects long-standing results in dataflow process networks, where bounded, local state supports deterministic and analyzable concurrent execution (Lee & Parks, Dataflow process networks, 1995; Dennis & Misunas, 1975). The internal representation (map, list, tree, etc.) is abstracted from the developer, ensuring backward compatibility and compiler freedom in optimization.

Crucially, node-state tables do not contradict POP’s immutability guarantees. Events remain immutable across the pipeline, and mutability is confined to node-local state, which is isolated, ephemeral, and analyzable at compile time. This mirrors broader results in stream processing systems, where limited mutable state can coexist with deterministic, replay-safe semantics when carefully constrained (Carbone, et al., 2015). The design also resonates with capability-based models (Levy, 1984), where access to resources is restricted to explicit, predefined operations, minimizing the attack surface and improving safety.

In this way, POP balances the global safety of immutability with a constrained, formally analyzable mechanism for stateful operations. Node-state tables give developers enough flexibility for state-dependent tasks while ensuring concurrency remains safe, observable, and predictable.

#### Determinism and Isolation

A critical property of concurrency in POP is that pipeline execution remains deterministic and isolated, even when entrypoints, workers and node-state tables are active. Determinism means that given the same inputs, a pipeline produces the same outputs regardless of scheduling order or parallelism level. This property underpins analyzability, reproducibility, and developer reasoning about correctness.

POP enforces determinism through three key mechanisms. First, immutable events ensure that no downstream worker can observe or mutate a value differently depending on execution order (§1.4.3.1). Second, node-state tables are strictly scoped: their contents are local to a node, ephemeral across program lifetimes, and protected by key-level locks (§1.4.3.2). This prevents concurrent workers from producing inconsistent results. Third, explicit concurrency semantics mean that the concurrency model is not left to runtime accident; instead, it is declared in pipeline structure and enforced by the compiler.

**Isolation complements determinism.** Each worker function executes inside a sandboxed virtual machine optimized to its declared capabilities. This ensures that side effects are bounded: workers cannot leak state across nodes, violate capability restrictions, or create hidden communication channels. The only permitted flows are via immutable events or node-local state tables, both of which are analyzable and observable.

This design resonates with established research in process networks and dataflow models, where determinism is preserved by enforcing functional purity across communication channels (Lee & Parks, 1995). It also echoes the principle of isolation in concurrent systems, where safe composition requires local reasoning and bounded side effects (Saltzer & Schroeder, 1975).

By combining immutability, scoped state, and sandboxing, POP ensures that concurrency yields performance and scalability without compromising analyzability. Developers can reason about pipelines in terms of event flow, confident that nondeterminism, race conditions, and state leakage are structurally excluded from the paradigm.

#### **Fault Containment and Recovery**

Concurrency in POP is designed not only for performance but also for resilience. In distributed and parallel systems, failures are inevitable: workers may crash, locks may time out, or resources may be exhausted (Beyer, Jones, Petoff, & Murphy, 2016). POP addresses these realities by embedding **fault containment and recovery mechanisms** directly into its concurrency model.

At the most basic level, **faults are contained within node boundaries**. Worker functions execute inside sandboxed virtual machines with strictly limited capabilities, so even if a worker encounters a runtime failure, the blast radius is confined. The failed worker’s event is wrapped in an Error<E> object and routed into the error-handling pipeline (§1.1.8), ensuring that faults are made explicit, analyzable, and recoverable.

Concurrency hazards such as deadlocks and livelocks are addressed by **timeouts and rollback semantics**. In node-state tables, transactions that cannot complete within the stateTableTombstone window are aborted, locks are released, and an error is emitted. This prevents stalled workers from indefinitely blocking progress, a common pitfall in shared-memory concurrency (Gray & Reuter, 1993). Similarly, backpressure policies (§1.1.7.3) ensure that when downstream nodes cannot keep up, events are either queued, dropped, or shunted in a manner that makes system behavior explicit rather than emergent.

Recovery in POP is facilitated by the ErrorPipeline(), which provides a structured mechanism for retries, logging, or alternate workflows. This approach resonates with fault-tolerant distributed dataflow systems such as Apache Flink, which embed recovery into the dataflow model rather than leaving it to ad hoc exception handling (Carbone et al., 2015). By combining immutability, isolation, and error pipelines, POP ensures that concurrency failures do not corrupt global state or compromise analyzability.

In summary, POP treats failures as first-class citizens of the concurrency model. By containing faults to nodes, enforcing rollback on stalled operations, and routing all errors through explicit pipelines, it avoids the silent corruption and unpredictable behavior that plague traditional concurrent systems (Lee, 2006). Fault containment and recovery are thus not optional add-ons but integral to POP’s concurrency safety.

### Worker Pools and Scheduling

Concurrency in Pipeline-Oriented Programming (POP) builds on two well-established traditions in computer science: **the actor model and dataflow process networks**. The actor model emphasizes lightweight, isolated workers that communicate exclusively through message passing, thereby avoiding the hazards of shared mutable state (Agha, 1986). Dataflow process networks, in turn, stress analyzability and determinism in concurrent systems, where computations are expressed as processes connected by immutable data channels (Lee & Parks, Dataflow process networks, 1995). POP synthesizes these approaches by assigning each pipeline node its own pool of workers, implemented as lightweight, sandboxed virtual machines.

Each worker pool executes node-specific logic in parallel, isolating failures and scaling elastically with demand. Importantly, not all workers are the same: some nodes—such as FanOut() and Collect()—have **compiler-generated workers**, derived entirely from declarative structure and optimized for analyzability. Other nodes—such as Ingress(), Transform(), and Egress()—host programmer-defined worker functions, which run within sandboxed environments and are restricted by declared capabilities. This distinction preserves both the formal guarantees of declarative structures and the flexibility of imperative computation where external interaction or custom logic is needed.

By coupling analyzability with elasticity, POP provides a concurrency model that avoids the pitfalls of nondeterministic threads while retaining adaptability under real-world workloads. Worker pools and scheduling are the mechanisms by which this synthesis becomes operational.

#### **Load Balancing**

Load balancing in POP emphasizes **lightweight process management** and predictable scheduling. Each node distributes incoming events across its pool of workers using strategies such as round-robin or least-loaded assignment. Unlike thread-based systems, where scheduling interleavings are opaque and nondeterministic, POP ensures that balancing policies are explicit, analyzable, and tied directly to node semantics. This follows the tradition of the actor model, where message delivery order and isolation define system behavior (Agha, 1986).

Because workers in POP are stateless by default, apart from controlled use of node-state tables, they can be created or retired with negligible overhead. This property allows load balancing to operate efficiently at fine granularity, without the synchronization and lifecycle costs associated with threads and shared memory (Lee, The problem with threads, 2006). Moreover, the runtime integrates load balancing with **elastic concurrency** (§1.4.4.3): when queues grow beyond thresholds, new workers are spawned; when utilization falls, workers are retired. This feedback loop ensures that balancing demand also drives dynamic scalability.

Finally, POP’s load balancing accounts for the distinction between **compiler-generated workers** and **programmer-defined workers**. Compiler-generated workers, such as those in FanOut() and Collect(), can be optimized aggressively and scale with near-zero overhead. Programmer-defined workers, used in Ingress(), Transform(), or Egress(), provide greater flexibility but may require more conservative scaling policies to preserve analyzability. By treating these two cases differently, POP achieves both efficiency and expressiveness.

#### Fault-Tolerant Scheduling

POP’s scheduler is designed so that **failures are contained, progress is maintained, and causes are observable** even under heavy concurrency. The design synthesizes two proven traditions: the **actor model’s** isolation and “let-it-crash” supervision philosophy (Agha, 1986; Armstrong, 2003), and **dataflow process networks’** analyzable, channel-based concurrency (Lee & Parks, Dataflow process networks, 1995). POP extends these with language-level guarantees—immutable Event<T>, capability-bounded sandboxes, and explicit error pipelines—that keep failure handling predictable and auditable.

**Isolation and failure domains.** Each node runs a pool of lightweight, sandboxed workers (cf. §1.4.4.1). A worker crash, panic, deadline overrun, or capability violation is **contained** to that worker. The in-flight event is wrapped as Error<E> with diagnostic metadata and routed to ErrorPipeline(); the failed worker is torn down and replaced. This mirrors actor supervision trees (Armstrong, 2003) while preserving dataflow analyzability (Lee & Parks, Dataflow process networks, 1995).

**Deadlines, timeouts, and backoff.** POP scheduling is deadline-aware: every worker invocation inherits node-level budgets (e.g., execution time, memory) and enforces operation timeouts—including stateTableTombstone for node-state transactions (§1.4.3.2). Exceeded budgets trigger rollback and error emission rather than silent stalls, aligning with SRE guidance that timeouts and bounded retries are necessary to avoid cascading failure (Beyer et al., 2016). Retries (if any) are governed by the pipeline’s error policy and by the backpressure regime (block → at-least-once; drop/shunt → at-most-once; cf. §1.1.7.3).

**Fairness and progress.** The scheduler prevents starvation via fair dispatch (e.g., round-robin or least-loaded) and by bounding long-running invocations with preemption via timeouts. Because events are immutable and workers are side-effect free by default, scheduler decisions do not compromise correctness—an advantage over shared-memory threads, where interleavings easily induce races and heisenbugs (Lee, The problem with threads, 2006).

**Relation to Go-style concurrency.** POP borrows from Go’s **CSP-inspired** model—fan-out/fan-in across lightweight execution contexts and explicit cancellation—while lifting these patterns to the language/compile-time level. Go encourages composing goroutines+ channels with context for deadlines and cancellation (Pike, 2012; Donovan & Kernighan, 2015). POP generalizes this: typed pipeline edges supplant ad hoc channels; backpressure and deadlines are declarative; and failure routes through an explicit error pipeline rather than ad hoc panic/recover. The result is Go-like operational simplicity with static analyzability akin to dataflow systems.

**Elastic fault response.** Scheduling is integrated with elastic concurrency (§1.4.4.3). If failure rates spike or queues deepen, the runtime can scale out workers (where safe) or scale in to relieve pressure, guided by observed service times and error budgets (Beyer, Jones, Petoff, & Murphy, 2016; Carbone, et al., 2015). Because workers are capability-bounded sandboxes, scaling changes do not enlarge the trust surface.

Taken together, POP’s fault-tolerant scheduling adheres to three invariants: (1) **containment**—failures don’t escape their worker/node; (2) **boundedness**—no unbounded waits or retries; and (3) **observability**—every failure becomes structured telemetry (Error<E> with provenance), enabling post-hoc analysis and targeted remediation.

#### Elastic Concurrency

Elastic concurrency in POP refers to the ability of each node’s worker pool to scale up or down dynamically in response to load, while preserving analyzability and fault containment. This concept extends beyond conventional multithreading, drawing inspiration from the actor model (Agha, 1986), dataflow networks (Donovan & Kernighan, 2015), and practical concurrency frameworks such as Erlang and Go (Armstrong, 2003; Donovan & Kernighan, 2015).

In actor systems such as Erlang, elasticity is achieved by spawning lightweight processes and supervising them hierarchically. When load increases, the system can scale by creating new actors, while failures are isolated and managed through “let it crash” semantics (Armstrong, 2003). POP borrows this principle but integrates it into the pipeline grammar: every node’s pool of workers is elastic, capable of expanding under backpressure and contracting under low utilization. Unlike ad hoc actor frameworks, these elasticity rules are analyzable at compile time, making them predictable and enforceable rather than emergent.

Similarly, Go’s concurrency model emphasizes lightweight goroutines and channel-based fan-out/fan-in, where millions of concurrent execution contexts can be scheduled efficiently (Donovan & Kernighan, 2015; Pike, 2012). POP generalizes this approach by binding elasticity to declarative node definitions. For example, FanOut() and Collect() nodes generate compiler-optimized workers that can scale aggressively, while Ingress(), Transform(), and Egress() nodes host programmer-defined workers whose elasticity is more conservative to preserve analyzability. This distinction ensures that POP can scale without sacrificing safety or semantic clarity.

Elastic concurrency also mirrors the behavior of distributed stream processing engines such as Apache Flink, where operator parallelism can grow or shrink dynamically to sustain throughput and latency under varying workloads (Carbone, et al., 2015). POP extends this practice into the language itself: elasticity is not left to deployment configuration or external orchestration but is expressed as part of the pipeline’s concurrency semantics.

The benefits of elastic concurrency are threefold. First, it allows POP programs to adapt gracefully to bursty or unpredictable workloads without manual intervention. Second, it preserves **fault containment** by ensuring that elasticity does not expand the trust boundary—new workers inherit the same sandbox and capability restrictions as existing ones. Third, it keeps concurrency analyzable: developers can reason about how pools expand or contract because scaling rules are visible in the declarative pipeline graph.

#### Summarizing POP Concurrency

The concurrency and scheduling strategies in Pipeline-Oriented Programming (POP) deliberately integrate principles from the **actor model** (Agha, 1986), **dataflow systems** (Lee & Parks, Dataflow process networks, 1995), and **lightweight concurrency frameworks** such as Go (Pike, 2012; Donovan & Kernighan, 2015). By grounding load balancing, fault tolerance, and elasticity in a declarative grammar, POP provides developers with concurrency primitives that are analyzable at compile time, yet adaptive at runtime.

Load balancing (Section 1.4.4.1) is achieved through lightweight, scalable worker pools that distribute events across multiple execution contexts with minimal overhead. Fault-tolerant scheduling (Section 1.4.4.2) ensures that worker failures do not compromise the correctness or liveness of the system, drawing on lessons from Erlang’s supervision hierarchies and Go’s recoverable goroutines. Elastic concurrency (Section 1.4.4.3) extends these strategies by dynamically scaling worker pools in response to load, combining adaptability with strict analyzability through compile-time rules.

Together, these mechanisms embody the paradigm’s guiding principles: concurrency safety through immutability, formal analyzability of event flow, and operational robustness in the face of uncertainty. POP’s contribution lies in integrating these established ideas into a single language-level abstraction. Where actor systems leave elasticity to runtime, and dataflow models sometimes struggle with fault recovery, POP unifies these concerns into a cohesive concurrency model.

The result is a paradigm that balances theory and practice: retaining the clarity of declarative graph structures while enabling the efficiency and resilience demanded by real-world systems. In this respect, POP demonstrates that concurrency and scheduling can be both **predictable** and **adaptive**, ensuring pipelines remain correct, safe, and efficient under diverse operating conditions.

### Error-Aware Scheduling

Error-aware scheduling in Pipeline-Oriented Programming (POP) integrates concurrency management with structured error handling. Unlike conventional models where errors are propagated through exceptions or ad hoc callbacks, POP treats errors as first-class events (Error<E>) that move through explicit pipelines. This design embeds retry logic, containment, and observability into the scheduling model itself, ensuring that failures are both analyzable and recoverable.

#### **Errors as First-Class Events**

All failures in POP are encapsulated as Error<E> objects, which inherit the metadata of the originating event (Event<T>) and augment it with error-specific diagnostics. The scheduler routes these errors through the error-handling pipeline (§1.1.8), allowing them to be analyzed, retried, or escalated systematically. This contrasts with exception-driven models in imperative languages, where errors can escape concurrency boundaries and cause nondeterministic failures (Scott, 2016). By treating errors as events, POP aligns with principles of event-driven reliability found in both the actor model (Agha, 1986) and stream-processing systems (Carbone, et al., 2015).

#### **Retry-Aware Scheduling**

POP allows bounded retries for failed worker functions, governed by compile-time or runtime policies. Retries are performed with explicit timeouts and backoff strategies, ensuring system stability even under persistent fault conditions. This approach parallels Erlang’s “let it crash” supervision trees (Armstrong, 2003)where failure recovery is systematic rather than ad hoc. Like Go’s goroutines with structured recovery (Donovan & Kernighan, 2015), POP’s retry-aware scheduling preserves liveness without sacrificing analyzability.

#### **Isolation and Containment**

Worker functions execute in sandboxed virtual machines constrained by declared capabilities. If a worker fails—whether due to runtime error, resource exhaustion, or external fault—the failure is **contained** to that worker and its current event. Node-state tables reinforce safety by enforcing atomicity: .set() and .update() operations roll back on failure (§1.4.3.2). This ensures that concurrency remains analyzable and no partial state corrupts pipeline execution, echoing actor-style process isolation (Agha, 1986) while extending it with explicit rollback semantics.

#### **Adaptive Backpressure with Error Awareness**

POP integrates error rates into its backpressure policies. If a node repeatedly produces errors, the scheduler adapts by shifting policy (e.g., from block to shunt), throttling workloads, or redirecting faulty streams into dedicated error paths. This ensures graceful degradation rather than cascading collapse, aligning with adaptive scheduling in dataflow networks (Lee & Parks, Dataflow process networks, 1995) and operational resilience principles emphasized in Site Reliability Engineering (Beyer, Jones, Petoff, & Murphy, 2016).

#### **Integration with Observability**

Each Error<E> carries an EventId, worker identity, node context, and failure classification (§1.1.8.2). This metadata makes failures observable as structured telemetry, supporting both real-time diagnosis and postmortem analysis. Unlike runtime-only monitoring, POP enforces observability as part of its semantics. This approach echoes operational best practices articulated in Google’s SRE discipline: “hope is not a strategy—errors must be measured, surfaced, and acted upon” (Beyer, Jones, Petoff, & Murphy, 2016).

#### **Summarizing Error-Aware Scheduling**

Error-aware scheduling in POP weaves error handling into the concurrency fabric itself. By treating errors as events, bounding retries, enforcing isolation, adapting backpressure, and mandating observability, POP ensures that pipelines are not only analyzable and performant but also robust in the face of inevitable failures. This synthesis advances beyond actor supervision or streaming fault-tolerance by embedding resilience directly into the language-level concurrency model.

### Scheduling Policies and Guarantees

POP provides analyzable concurrency by making scheduling policies explicit and binding them to the pipeline grammar. Where traditional thread libraries and even actor frameworks often leave scheduling to runtime heuristics, POP defines scheduling as a **compile-time property** with predictable guarantees.

#### **Deterministic vs. Nondeterministic Scheduling**

Scheduling in Pipeline-Oriented Programming (POP) operates on a spectrum between determinism and nondeterminism. Deterministic scheduling refers to execution orders that can be fully resolved at compile time, while nondeterministic scheduling depends on dynamic runtime conditions such as external inputs or conditional branching. POP leverages its declarative pipeline structure to maximize determinism where possible, while confining nondeterminism to explicit, analyzable boundaries.

In **dataflow systems**, deterministic scheduling is feasible when node firing rules are data-independent. For example, if every invocation of a Transform() node consumes exactly one event and produces exactly one event, then a static schedule can be derived in advance. Lee and Parks (1995) showed that in such cases, schedulers can guarantee deadlock-freedom, bounded buffer sizes, and efficient execution with minimal runtime overhead. POP adopts this principle: whenever pipeline segments exhibit predictable consumption and production rates, the compiler can resolve scheduling decisions at compile time.

However, real-world systems inevitably involve nondeterministic inputs and dynamic control flow. External events—such as network messages, user input, or file reads—arrive unpredictably. Conditional branches in FanOut() nodes and variable mappings in Transform() nodes also prevent a purely static schedule. In these cases, POP relies on a runtime scheduler that dispatches events dynamically across worker pools. To mitigate uncertainty, the runtime is still constrained by compile-time declarations such as backpressure policies, deadlines, and worker pool bounds. This ensures that nondeterminism does not spread unchecked but remains confined to ingress points and conditional structures.

The advantage of this hybrid model is that POP can provide **stronger guarantees than purely dynamic systems**. For statically analyzable segments, it ensures deadlock freedom, bounded latency, and predictable resource use. For dynamic segments, it enforces safety through explicit runtime policies rather than relying on emergent thread interleavings. This approach balances the efficiency of deterministic scheduling with the flexibility of dynamic concurrency.

From a historical perspective, POP’s design draws from multiple traditions. Dataflow scheduling research highlights the benefits of static analysis where possible (Lee & Parks, Dataflow process networks, 1995). Real-time scheduling theory demonstrates how deadlines and rate-monotonic analysis can guarantee responsiveness in predictable workloads (Liu & Layland, 1973). Meanwhile, lightweight concurrency models such as Go’s goroutines emphasize efficient dynamic scheduling of millions of concurrent tasks (Pike, 2012). By integrating these insights, POP offers a concurrency model that is analyzable, predictable, and adaptive, minimizing runtime uncertainty without sacrificing expressiveness.

#### **Priority and Fairness**

A central challenge of concurrent systems is ensuring that event scheduling remains both fair and responsive. Traditional thread-based environments often leave scheduling to the operating system, resulting in nondeterministic interleavings that can cause starvation or unpredictable latency (Lee, The problem with threads, 2006). POP addresses this by embedding **priority and fairness policies** directly into the pipeline grammar, making them analyzable at compile time and enforceable at runtime.

**Fairness** in POP ensures that no event flow can be indefinitely starved by others. Worker pools within each node dispatch events using explicit policies, such as round-robin or least-loaded assignment. Because these policies are declared as part of the node’s configuration, their behavior is transparent to both developers and the compiler. This differs from ad hoc thread pools, where fairness guarantees depend on low-level runtime heuristics and can vary across platforms (Scott, 2016).

**Priority**, in contrast, allows certain event flows to receive preferential service. For example, telemetry or safety-critical events may be marked with higher priority to guarantee low-latency processing. POP ensures that priority is not emergent or accidental but declared as part of the pipeline’s semantics. This reflects lessons from real-time scheduling theory, where rate-monotonic and deadline-based policies are used to ensure responsiveness under load (Liu & Layland, 1973). By binding such priorities to the pipeline, POP provides developers with predictable guarantees about latency and throughput.

This integration of fairness and priority reflects the paradigm’s hybrid philosophy: borrowing from real-time systems the ability to provide hard guarantees, while retaining the scalability and elasticity of modern actor and dataflow models. The result is that POP pipelines can achieve **predictable responsiveness** under contention, without sacrificing the analyzability that makes compile-time reasoning possible.

#### Latency and Throughput Guarantees

Figure 9: Concurrency Balancing Problem

A black triangle on a white background

AI-generated content may be incorrect.

Concurrency models must balance latency—the time to process individual events—with throughput—the number of events handled per unit of time. Traditional thread-based systems often make these properties emergent outcomes of runtime scheduling, which can lead to unpredictable behavior under load (Lee, The problem with threads, 2006). In contrast, POP binds latency and throughput guarantee directly to the pipeline graph, ensuring that they are analyzable at compile time and enforceable at runtime.

POP provides latency guarantees by allowing nodes to declare deadlines and executionbudgets. For example, an Ingress() node may specify a maximum response time for reading external data, while a Transform() node can enforce per-event execution timeouts. These budgets resemble concepts from real-time systems theory, such as deadline-monotonic and rate-monotonic scheduling, which guarantee responsiveness under predictable workloads (Liu & Layland, 1973). When deadlines are missed, events are wrapped as Error<E> objects and routed into error pipelines (§1.1.8), preserving observability rather than silently degrading performance.

Throughput guarantees are achieved by integrating scheduling with backpressurepolicies (§1.1.7.3). A block policy ensures at-least-once delivery by pausing upstream emission, while dropOldest, dropNewest, or shunt policies enforce bounded throughput by discarding or rerouting excess events. Because these policies are explicit in the pipeline, throughput is predictable and analyzable rather than left to runtime heuristics.

Crucially, POP ensures that latency and throughput policies are not competing concerns but coordinated aspects of the concurrency model. For example, the runtime may scale worker pools elastically (§1.4.4.3) when queues grow beyond thresholds, improving throughput without violating latency constraints. Compiler analysis also enables static optimization of worker placement for predictable workloads, minimizing scheduling overhead in segments where event production and consumption rates are known (Lee & Parks, Dataflow process networks, 1995).

This explicit integration of latency and throughput guarantees situates POP between hard real-time systems, which require strict deadline adherence, and best-effortstreaming frameworks, which often favor throughput over latency. POP provides boundedguarantees: deadlines and throughput constraints are enforced declaratively, violations are observable, and trade-offs are transparent to developers. The result is a model that balances efficiency, predictability, and operational resilience.

#### Integration with Error-Aware Scheduling

Scheduling in POP is not only a matter of distributing events efficiently across workers but also of ensuring resilience when failures occur. To this end, POP integrates its scheduling policies with error-aware scheduling (§1.4.5), creating a unified model in which both normal (Event<T>) and error events (Error<E>) are treated as first-class citizens within the concurrency framework. This integration ensures that latency, throughput, and fairness guarantee extend beyond normal execution flows to include error handling and recovery.

First, POP enforces bounded retries within its scheduling policies. Workers that fail are retried only under declared constraints—such as maximum retry counts, deadlines, or backoff—preventing unbounded resource consumption. This approach reflects the supervision principles of the actor model, where errors are contained and managed by structured policies rather than left to emergent behavior (Agha, 1986; Armstrong, 2003). By embedding these retry limits into the scheduler, POP ensures that fault recovery remains predictable and analyzable.

Second, POP guarantees isolation between error traffic and normal traffic. In traditional concurrent systems, bursts of errors can overwhelm resources, starving legitimate workloads and degrading system responsiveness. POP addresses this by maintaining separate error queues with their own scheduling policies. This practice mirrors techniques in real-time scheduling, where tasks are partitioned by criticality to avoid starvation and maintain latency guarantees (Liu & Layland, 1973). Errors are thus observable and recoverable without jeopardizing the progress of the main pipeline.

Third, POP couples its backpressuremechanisms with error handling. If error rates exceed thresholds, the scheduler adapts dynamically, throttling or rerouting workloads to prevent cascading failure. This design resonates with adaptive backpressure control in dataflow systems, which balance throughput and stability under varying load conditions (Lee & Parks, Dataflow process networks, 1995). By making this coupling explicit in the pipeline grammar, POP ensures that error-related congestion is visible and analyzable at compile time, not an emergent runtime phenomenon.

Finally, POP integrates observabilityintoschedulingdecisions. Every Error<E> object carries structured metadata, including its EventId, worker identity, and failure classification (§1.1.8.2). This metadata feeds back into the scheduler, enabling adaptive decisions such as rebalancing worker pools or escalating failures to error pipelines. This aligns with modern operational guidance from Site Reliability Engineering (SRE), which emphasizes that system health depends on making failures observable and actionable rather than suppressing them (Beyer, Jones, Petoff, & Murphy, 2016).

By weaving error-aware semantics into its scheduling framework, POP extends fairness, latency, and throughput guarantees into the domain of fault tolerance. The result is a concurrency model that does not treat errors as exceptions to be bolted on after the fact but as integral events subject to the same analyzable and predictable policies as all other computation.

#### **Summarizing Scheduling Policies and Guarantees**

POP’s scheduling model synthesizes insights from dataflow networks, real-time systems, and lightweight concurrency frameworks into a coherent, analyzable whole. By distinguishing between **deterministic** and **nondeterministic** scheduling, POP makes explicit which pipeline segments can be resolved at compile time and which require runtime adaptation (§1.4.6.1). This ensures that deadlock-freedom, bounded latency, and resource constraints are not emergent properties but formally analyzable guarantees.

**Fairness and priority** are built into the pipeline grammar (§1.4.6.2), preventing starvation while enabling preferential treatment for latency-sensitive flows. **Latency and throughput guarantees** are expressed declaratively, enforced through deadlines and backpressure policies, and supported by elastic concurrency strategies that preserve both efficiency and predictability (§1.4.6.3). Importantly, these guarantees extend into **error-aware scheduling** (§1.4.6.4), ensuring that fault recovery does not undermine fairness or throughput and that errors remain observable, bounded, and analyzable.

Together, these policies demonstrate POP’s central contribution: **scheduling is not a hidden runtime mechanism but a language-level property**. By exposing and constraining scheduling semantics at compile time, POP reduces runtime uncertainty, unifies correctness with operational safety, and provides developers with a model of concurrency that is both **predictable in theory and robust in practice**.

## Worker Functions

Worker functions are the imperative atoms of Pipeline-Oriented Programming (POP): they execute within node-scoped worker pools, transform immutable Event<T> objects, and interact with the outside world only where explicitly permitted. POP constrains workers by design—purity by default, sandboxed execution, explicit capabilities, and analyzable concurrency—to preserve the global guarantees of the declarative pipeline (analyzability, safety, observability) while retaining sufficient expressiveness for real-world systems (Agha, 1986; Lee & Parks, Dataflow process networks, 1995; Saltzer & Schroeder, The protection of information in computer systems, 1975).

### Design Principles

Worker functions in Pipeline-Oriented Programming (POP) embody the bridge between declarative pipeline graphs and imperative computation. Their design is governed by principles that prioritize analyzability, concurrency safety, and operational resilience, while still permitting sufficient expressiveness for real-world use. These principles draw on decades of work in dataflow (Dennis & Misunas, 1975; Lee & Parks, Dataflow process networks, 1995), capability-based security (Levy, 1984), and programming language design (Scott, 2016), as well as modern systems practice (Beyer, Jones, Petoff, & Murphy, 2016).

#### Purity by Default

Worker functions are pure by default: they treat their inputs (Event<T>) as strongly typed and immutable and yield new values rather than mutating existing ones. This echoes the guarantees of functional programming and dataflow architectures, where immutability enables deterministic analysis and eliminates many classes of race conditions (Lee & Parks, Dataflow process networks, 1995; Peyton Jones, 2003). By requiring purity as the default discipline, POP ensures that pipelines remain analyzable, predictable, and safe to parallelize.

#### Sandboxed Minimalism

Every worker function executes inside a compiler-optimized sandboxed virtual machine that includes only the minimal feature set required. This sandbox is configured at compile time to respect the node’s declared capabilities (e.g., FileRead, NetIn) and runtime modifiers (e.g., FileRead:/opt/\*.json) (Levy, 1984). This principle reflects Saltzer and Schroeder’s (1975) least privilege rule: workers are incapable of accessing undeclared resources, thus reducing attack surfaces and bounding blast radius in the event of compromise.

#### Analyzable Concurrency

Concurrency in POP emerges from the independent execution of workers across events, rather than shared-memory threads. The design follows the actor model (Agha, 1986) and statically analyzable dataflow networks (Lee & Parks, Dataflow process networks, 1995), ensuring that concurrency arises from composition rather than uncontrolled interleaving. POP extends this tradition by making concurrency guarantees explicit at the language level, not emergent from runtime configuration.

#### Errors as First-Class Events

A central principle of POP is that failures are not side effects but first-class events: any violation of capability, type, or runtime constraint produces an Error<E> object, preserving the provenance of the originating event. These errors are routed into explicit error pipelines for handling, retry, or escalation. This design aligns with distributed systems practice, where structured error-handling and observability are critical for reliability (Carbone, et al., 2015; Beyer, Jones, Petoff, & Murphy, 2016; Kleppmann, 2017). By treating errors as analyzable objects, POP avoids silent failures and enforces correctness contracts throughout the pipeline lifecycle.

### Execution Model and Sandbox

Worker functions in Pipeline-Oriented Programming (POP) execute inside **compiler-synthesized, capability-bounded sandboxes**. The sandbox makes the worker’s effects **explicit, minimal, and analyzable**: instruction sets and system interfaces are narrowed to what the program declares at compile time; runtime enforcers meter time, memory, and I/O; and any violation is surfaced as Error<E> with full provenance (§1.1.8). By constraining effects at the execution boundary rather than by convention, POP operationalizes classic principles of **least privilege** and **capability security** (Levy, 1984; Saltzer & Schroeder, The protection of information in computer systems, 1975), while preserving the compile-time reasoning benefits of dataflow (Lee & Parks, Dataflow process networks, 1995).

#### Worker Lifecycle and Isolation

Each pipeline node owns a **pool of lightweight workers** (§1.4.4). A worker is instantiated as a sandboxed virtual machine that loads a single worker function and runs to completion for an event, after which it is recycled or retired. Isolation is **per worker, per event**; memory is zeroed or discarded between invocations; there is no shared mutable state across workers or nodes; and inter-worker coordination, if needed, is restricted to **node-state tables** (§1.4.3.2). This lifecycle enforces **fault containment**—crashes, panics, or timeouts are confined to the current event and worker, and are converted to Error<E> for explicit handling (§1.4.5).

#### Capability Enforcement (Compile-Time and Runtime)

All external effects are mediated by capabilities declared at compile time (e.g., FileRead, DirectoryList, NetIn), optionally refined by **runtime modifiers** (e.g., FileRead:/opt/\*.json, NetIn:127.0.0.1:8080). The compiler **specializes** the sandbox to include only the minimal syscalls and libraries needed to honor these declarations; the runtime **enforces** them through system call interposition and descriptors that cannot be forged or escalated. Any attempt to exceed the declared authority results in immediate termination of the operation and emission of Error<E>. This end-to-end design is a direct application of **capability-based computer systems** and **least privilege** (Levy, 1984; Saltzer & Schroeder, The protection of information in computer systems, 1975).

#### I/O Discipline by Node Role

POP **centralizes I/O** in Ingress() and Egress() workers. Transform() workers are **I/O-free** and operate purely on their inputs, preserving analyzability and enabling aggressive optimization. This **role-segregated I/O** collapses a large class of correctness and security risks—implicit effects and covert channels—into two analyzable boundaries (Levy, 1984; Saltzer & Schroeder, The protection of information in computer systems, 1975). When I/O occurs, it must be **explicitly capability-authorized** and is automatically annotated with event metadata for observability (§1.1.6.3; 1.6).

#### Resource Budgets, Deadlines, and Timeouts

Each worker invocation carries bounded resource budgets (CPU‐time, memory, descriptors) and deadlines that the scheduler enforces (§1.4.6.3). Node-state table operations (set, update) are atomic with key-level locking and a compile-time stateTableTombstone timeout; overruns roll back and emit Error<E> (§1.4.3.2). For I/O, timeouts and bounded retries prevent retry storms and cascading failure, aligning with site reliability guidance that faults must be bounded and (Beyer, Jones, Petoff, & Murphy, 2016). These controls make progress guarantees and failure modes explicit rather than emergent.

#### Observability Hooks and Provenance

The sandbox **attaches structured telemetry** to every invocation: EventId, nodeId, workerId, timing (queueing, service, total), capability usage, and error classification when applicable. Because events are immutable and carry causal metadata, distributed traces can be reconstructed deterministically without developer-authored ad hoc logging (§1.1.6.3; §1.4.5). Errors become **first-class events** routed through ErrorPipeline() with full provenance (Carbone, et al., 2015; Beyer, Jones, Petoff, & Murphy, 2016).

#### Compiler Integration and Specialization

Worker functions are compiled to an intermediate representation to enable **whole-pipeline optimization** (e.g., inlining, copy-elision, fusion) while preserving semantic boundaries for I/O and mutability. POP’s use of an IR such as **LLVM** supports **target-independent optimization** **and late specialization** of sandboxed runtimes (Lattner & Adve, 2004). Declarative nodes (FanOut(), Collect()) become **compile-generated workers** that admit stronger static scheduling (§1.4.6.1), whereas Ingress(), Transform(), and Egress() host **programmer-defined** workers whose performance is improved by IR-level optimization under the same sandbox constraints.

#### Concurrency Inside the Sandbox

Workers may spawn **local, capability-bounded tasks** to overlap computation (e.g., parsing + hashing) so long as they remain within the same sandbox, share no global mutable state, and communicate through **typed local queues**. This achieves Go-like ergonomics for internal concurrency while preserving POP’s compile-time analyzability and isolation (Pike, 2012; Donovan & Kernighan, 2015).

### Function Categories

Functions in Pipeline-Oriented Programming (POP) are designed with a tightly constrained interface for workers and a flexible but analyzable structure for helpers. This division balances the need for **predictable analyzability** in the pipeline graph with the **expressiveness and modularity** required for real-world computation.

#### **Worker Functions**

In AMI, a worker function bound to a node (Ingress(), Transform(), Egress()) is written with a concrete surface form:

Figure 10: Worker Function Signature

|  |
| --- |
| func <funcIdentifier> (in Event<T>) (out Event<U>, err Error<E>) |

This is an example of the abstract form all worker functions must to be compatible with the POP standard. It implements POP’s **uniform worker interface**. In addition to the typing and sandbox rules already stated, POP defines precise semantics for **nil/none** results:

* **Success, no downstream emission**: out == nil and err == nil

The worker intentionally emitsnothing (e.g., a filter that drops events not meeting a predicate). No message is enqueued on the downstream edge and **no error** is routed to ErrorPipeline(). Counters/telemetry record the drop as a normal, non-error outcome (cf. §1.5.2.5, §1.6).

* **Success with emission**: out != nil and err == nil

A single Event<U> is produced and forwarded. This is the ordinary success path.

* **Failure with diagnostics**: out == nil and err != nil  
  No downstream event is produced; the original Event<T> is wrapped as Error<E> and routed into the error-handling pipeline (cf. §1.1.8).
* **Forbidden:** out != nil and err != nil

Producing both would violate analyzability and delivery accounting; the compiler rejects such workers.

These rules preserve POP’s **single-disposition** property per invocation (exactly one of: emit, error, or silence), which keeps scheduling, backpressure, and observability predictable and analyzable (Lee & Parks, Dataflow process networks, 1995; Scott, 2016). Transform workers commonly use the “silent success” case to implement filtering; ingress and egress workers typically prefer explicit emissions or explicit errors, but may also return silence when policies or capabilities dictate (e.g., rate-limited probe, heartbeat tick with no data).

As before, workers remain pure by default and run in a capability-bounded sandbox; allowing nil for out or err does not change those constraints—only the worker’s disposition for the current Event<T>.

### Purity, Immutability, and Controlled Mutability

One of the defining features of Pipeline-Oriented Programming (POP) is its treatment of function purity and immutability. Worker and helper functions are designed to be analyzable and deterministic by default, which requires strict enforcement of immutability across events and data structures. Yet real-world computation occasionally requires local mutation—for example, incrementing a counter or buffering a temporary structure. POP addresses this through the mutable()operator, a tightly scoped construct that introduces controlled, analyzable mutability.

By embedding these constraints directly into the function model, POP inherits the benefits of functional programming’s immutability guarantees (Peyton Jones, 2003), while enabling limited state changes in the manner of structured dataflow systems (Dennis & Misunas, 1975; Lee & Parks, Dataflow process networks, 1995).

#### **Purity as the Default Rule**

In POP, all functions are **pure by default**:

* Their outputs depend solely on their inputs.
* They perform no I/O except where explicitly authorized through declared capabilities.
* They cannot mutate state outside their own scope.

This ensures that pipelines are predictable, reproducible, and analyzable at compile time, eliminating hidden side effects that complicate reasoning in traditional event-driven systems (Scott, 2016). Purity also allows the compiler to apply optimizations such as function inlining, constant propagation, and dead-code elimination safely (Lattner & Adve, 2004).

#### **The** mutable() **Operator**

For cases where local mutation is necessary, POP provides the mutable() **operator**, which:

* Creates a tightly scoped region where temporary mutation is permitted, protected by transparent locks.
* Requires compile-time validation: the compiler ensures mutations do not escape the scope or leak across worker or node boundaries.
* Uses rollback semantics if an error occurs, ensuring consistency with event-level atomicity.

Example use cases include:

* Incrementally building a local buffer.
* Updating a scratchpad for intermediate calculations.
* Caching repeated sub-computations during one worker invocation.

The operator is analogous to **linear types** and controlled effect systems in functional programming (Peyton Jones, 2003), enabling pragmatic mutation without compromising analyzability.

#### **Enforcement and Guarantees**

POP enforces immutability and scoped mutability through:

1. **Compile-time checks** that ensure mutable() is used only within valid scopes.
2. **Runtime guards** in the sandbox (§1.5.2) that prevent unauthorized state changes.
3. **Type-level guarantees**: any function using mutable() has its effect annotated in its signature, making mutability visible to the compiler and to developers.

These rules ensure that even when mutability is used, it remains analyzable, explicit, and confined to the smallest possible scope.

#### **Implications for Concurrency and Optimization**

By treating purity as the norm and restricting mutability to well-defined blocks, POP combines the strengths of both functional and dataflow programming:

* Purity and immutability allow strong static analysis and aggressive compiler optimization.
* Controlled mutability enables performance-sensitive computations without sacrificing analyzability or safety.
* The compiler can reason about scheduling and backpressure policies (§1.4.6) without uncertainty introduced by hidden shared state.

This dual approach aligns POP with modern research that balances purity with practical needs in concurrent and distributed systems (Lee & Parks, Dataflow process networks, 1995; Peyton Jones, 2003; Scott, 2016).

### Function Reuse and Composition

While POP constrains worker functions to a uniform signature (§1.5.3) to ensure analyzability, the paradigm recognizes that practical programming requires **function reuse and modularity**. Helper functions enable developers to abstract repeated logic, improve maintainability, and avoid duplication, while still remaining analyzable within POP’s strict type and purity discipline. By supporting both **reusable general functions** and **inline anonymous functions**, POP balances the rigor of strongly typed declarative pipelines with the flexibility of conventional software engineering practices (Peyton Jones, 2003; Scott, 2016).

#### **Reusable General Functions**

Reusable general functions are **named helper functions** intended to be invoked across multiple workers and pipelines. They encapsulate common logic—for example, parsing, validation, or encoding—while adhering to POP’s rules of purity and immutability. Unlike worker functions, general functions do not need to follow the uniform Event<T> → Event<U> | Error<E> interface. Instead, they may define arbitrary type-safe signatures, provided they:

1. Remain **pure** by default, with side effects only allowed inside mutable() blocks.
2. Are strongly typed, such that inputs and outputs can be fully checked at compile time.
3. Do not perform I/O or escape the sandbox; all capability-bound operations must occur within workers (§1.5.2).

This allows general functions to act as **building blocks**, while workers remain the **structural joints** of the pipeline. For instance, a parsing helper might be defined once and reused across multiple Ingress() workers, ensuring consistency and avoiding redundancy.

Compiler integration enhances reuse by supporting **inlining and specialization** at the intermediate representation (Lattner & Adve, 2004). This ensures that modularity does not impose runtime overhead and that reused functions remain analyzable. The approach reflects functional programming’s long-standing reliance on **higher-order abstractions** for reuse and modularity (Peyton Jones, 2003), but channels them into a **deterministic pipeline model**.

#### **Inline Anonymous Functions**

In addition to reusable named helpers (§1.5.5.1), POP supports **inline anonymous functions**, often referred to as lambdas. These functions provide a mechanism for embedding small, self-contained computations directly inside worker definitions without the overhead of naming or separately compiling them. Their role is pragmatic: they allow pipeline authors to implement **localized logic** such as filters, projections, or value transformations without breaking flow readability.

Like all functions in POP, anonymous functions remain subject to the same **purity and immutability rules**. They operate on immutable data, cannot perform I/O, and may only use the mutable() operator inside tightly scoped blocks (§1.5.4). This ensures that convenience never comes at the expense of analyzability. The compiler treats lambdas as first-class citizens in the type system, giving them explicit signatures based on their inputs and outputs, even when defined inline.

A key advantage of inline anonymous functions is their synergy with **compiler optimizations**. Since they are typically small and local, compilers can aggressively **inline and specialize** them, eliminating call overhead while preserving semantic guarantees (Lattner & Adve, 2004). This allows POP to support idiomatic, functional-style composition (Peyton Jones, 2003) without sacrificing runtime performance.

From a design standpoint, inline anonymous functions enhance **readability and expressiveness**, making it possible to keep transformation logic close to its point of use. At the same time, their constraints prevent them from introducing hidden side effects, ensuring they integrate smoothly into POP’s declarative pipeline graphs. Thus, POP offers the benefits of functional-style lambda expressions while maintaining a strict separation between **inline convenience and structural pipeline analyzability.**

#### Function Composition

Function composition in POP serves a narrow but essential purpose: it lets authors build worker logic from small, reusable, pure helper functions without eroding analyzability, safety, or performance. Composition occurs **inside** a worker’s sandbox (Ingress(), Transform(), or Egress()) and never crosses node boundaries. Cross-node composition remains a pipeline concern (cf. §1.3), whereas **function** composition is a local structuring device for clarity and reuse within the worker’s single-disposition contract (exactly one of: emit, error, or silence).

**Semantics and constraints.** POP adopts the standard, type-safe notion of composition—sequencing functions so the output type of one is the input type of the next—under a strict discipline: helpers are pure and effect-free (no I/O; no shared state), and any temporary mutation must be wrapped by mutable() within a narrow lexical scope (§1.5.4). These constraints preserve referential transparency for helpers and keep the composed worker analyzable and deterministic (Peyton Jones, 2003; Scott, 2016). Determinism at the pipeline level is retained because composed helpers execute within a single worker activation and communicate outward only via the worker’s Event/Error result, aligning with dataflow process-network results on determinate computation (Lee & Parks, Dataflow process networks, 1995).

**Compositional patterns.** In practice, workers apply small compositions such as *validate ∘ normalize ∘ transform*: e.g., a Boolean-valued validate(T), a shape-preserving normalize(T) → T, then a map(T) → U. On failure, the worker returns (out=nil, err=Error<E>); on a filtered path, (out=nil, err=nil); on success, (out=Event<U>, err=nil). Keeping error creation at the worker boundary (rather than in helpers) preserves POP’s single-disposition rule and makes error semantics explicit and analyzable (Scott, 2016; Lee & Parks, Dataflow process networks, 1995). Higher-order helpers are permitted: a helper may accept other pure helpers (e.g., a predicate or projector) to produce a specialized function, enabling library-quality reuse while preserving purity (Peyton Jones, 2003).

**Type safety and identity.** Because POP is strongly typed, legal compositions are enforced at compile time: the compiler verifies that helper output types align with downstream inputs and that the worker’s ultimate signature remains Event<T> → (Event<U> | Error<E> | silence). This mirrors classic type-theoretic discipline for modular, analyzable programs (Cardelli, Type Systems, 1989; Scott, 2016). Signatures determine identity: a change to a helper’s inputs/outputs is a breaking change that the compiler surfaces immediately, preventing schema drift.

**Optimization and cost.** Composition improves clarity without imposing runtime overhead. Pure, small helpers (including inline anonymous functions from §1.5.5.2) are excellent candidates for IR-level inlining, constant propagation, and copy elision under a modern toolchain (e.g., LLVM), yielding zero-cost abstractions in the common case (Lattner & Adve, 2004; Scott, 2016). Because helpers cannot perform I/O or access ambient capabilities, composition never widens the trust surface or the TCB of the worker sandbox, preserving least-privilege invariants (Levy, 1984) (Saltzer & Schroeder, The protection of information in computer systems, 1975).

**What composition does *not* do.** POP deliberately forbids composition that would blur boundaries: helpers cannot emit events, touch node-state tables, or perform I/O; workers cannot “compose” across nodes (that’s pipeline composition); and composed helpers cannot create implicit side channels. These constraints reflect lessons from dataflow and concurrent systems: correctness and predictability depend on isolating effects, bounding state, and keeping control/ data movement explicit (Lee & Parks, Dataflow process networks, 1995; Saltzer & Schroeder, The protection of information in computer systems, 1975).

**Summary.** POP’s function composition is an intentionally modest, locally scoped tool: chain small, pure helpers inside a worker to increase clarity and reuse; surface all effects at the worker boundary; and rely on the compiler to enforce type safety and erase abstraction overhead. The result is code that reads functionally, compiles predictably, schedules deterministically, and preserves the paradigm’s core guarantees of analyzability, security, and observability (Peyton Jones, 2003; Lee & Parks, Dataflow process networks, 1995; Lattner & Adve, 2004; Scott, 2016; Levy, 1984; Saltzer & Schroeder, The protection of information in computer systems, 1975).

#### Higher-Order Functions

Higher-order functions (HOFs) extend the expressive power of POP by allowing functions to take other functions as parameters or return them as values. While this idea originates in functional programming (Backus, 1978; Peyton Jones, 2003), POP adopts HOFs in a constrained manner that preserves analyzability, determinism, and type safety.

**Semantics in POP.** Within POP, HOFs are primarily used for helper functions rather than worker functions. For example, a map helper may accept a pure function f: T → U and apply it over event payloads before passing them into the worker’s pipeline logic. This allows concise expression of reusable transformations while respecting the worker’s strict Event<T> → (Event<U>, Error<E>) interface. Worker functions themselves are **not** higher-order: they cannot emit new functions at runtime or accept arbitrary executable code from outside the compiler’s analyzable scope.

**Safety constraints.** To ensure analyzability and security, POP restricts HOFs to pure functions with no external side effects or I/O. Their inputs and outputs must remain strongly typed, and the compiler validates all compositions at compile time, enforcing that a higher-order parameter function conforms to expected signatures. This discipline is consistent with type theory foundations of higher-order polymorphism (Cardelli & Wegner, 1985) and modern functional programming practices (Peyton Jones, 2003) (Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985).

**Benefits.** The use of HOFs brings three concrete advantages:

* **Reusability** — Generic transformations (e.g., predicates, mapping functions, reducers) can be shared across multiple workers without code duplication.
* **Clarity** — Logic such as filtering, projection, or enrichment can be expressed at the point of use without inlining repetitive code.
* **Optimization** — Pure HOFs are excellent candidates for inlining, partial evaluation, and fusion at the IR level, enabling the compiler to eliminate abstraction overhead (Lattner & Adve, 2004).

**Limitations.** POP deliberately avoids exposing runtime function construction or reflection through HOFs. While languages like JavaScript or Python allow arbitrary closures to be passed around dynamically, POP forbids such features to keep pipelines statically analyzable and to prevent injection or runtime ambiguity. This aligns with the paradigm’s commitment to immutability, strong typing, and compiler-enforced guarantees (Lee & Parks, Dataflow process networks, 1995).

**Summary.** Higher-order functions in POP are a disciplined tool: allowed for pure helpers, disallowed for workers, and always enforced through compile-time type checking. By restricting their scope, POP captures the benefits of functional abstraction while ensuring that pipelines remain analyzable, predictable, and safe (Backus, 1978; Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985; Peyton Jones, 2003; Lee & Parks, Dataflow process networks, 1995; Lattner & Adve, 2004)

#### Reusability Across Pipelines

Reusability is a foundational principle in modern programming paradigms, enabling developers to reduce redundancy, improve maintainability, and enforce consistent semantics across projects. In Pipeline-Oriented Programming (POP), reusability is achieved not only through modular pipeline design but also through the reuse of worker and helper functions across package and namespace boundaries. Functions can be invoked with fully-qualified names, ensuring that the compiler can resolve their scope unambiguously, regardless of the calling context.

This approach reflects long-standing principles of modular programming, where separation of concerns and clearly defined interfaces enable compositionality and reuse (Parnas, 1972; Meyer, 1997). By requiring strong typing and unambiguous function signatures, POP ensures that reusable components can be safely composed across multiple pipelines without introducing type mismatches or behavioral ambiguity. The binding of a function’s identity—its role (worker vs. helper), inputs, and outputs—into a unified signature allows the compiler to guarantee compatibility when functions are imported into different declarative graphs.

At the same time, POP extends this principle by making function reusability explicit at the pipeline level. A worker function defined in one package can be invoked by a pipeline in another package without compromising analyzability, as the compiler validates the function’s declared signature, capability constraints, and error contracts at compile time. This stands in contrast to many event-driven systems, where implicit dependencies or runtime discovery mechanisms can obscure the provenance and trustworthiness of external function (Schneider, 2000).

Reusability in POP also reinforces the paradigm’s emphasis on analyzability and immutability. Because worker functions are pure and constrained by their capabilities, reusing them across pipelines does not create hidden state dependencies or side effects that would undermine formal reasoning. Helper functions, while more flexible, remain bound by strong typing rules, ensuring that even user-defined abstractions can be reused without introducing unsafe polymorphism or type coercion (Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985).

Finally, function reusability across pipelines supports the ecosystem-level benefits of POP. Organizations can develop shared libraries of ingress handlers, transformations, and error processors, validated once and reused many times across domains. This parallels the rise of reusable service components in microservice architectures but operates at a finer granularity, at the level of typed event transformations rather than network services (Newman, 2015). By embedding reusability into the paradigm itself, POP ensures that pipelines scale not only in performance but also in maintainability and long-term evolution.

### Worker Function Execution Context

Worker functions in Pipeline-Oriented Programming (POP) represent the imperative boundary within otherwise declarative pipelines. While pipelines define event flow as analyzable graphs, worker functions provide the computational logic that processes individual events. Their execution context encompasses lifecycle management, scheduling, mutability, node-state interactions, and I/O boundaries. This section formalizes the principles governing worker execution, ensuring consistency with POP’s core goals of analyzability, concurrency safety, and observability.

#### Lifecycle and Scheduling

The lifecycle of a worker function begins when its host node receives an incoming Event<T> and ends with either the emission of a new event Event<U> or the wrapping of an error in Error<E>. Between these boundaries, execution proceeds within a compiler-optimized, sandboxed virtual machine restricted by the declared capabilities of the node. This ensures that each worker operates in a least-privilege environment (Saltzer & Schroeder, The protection of information in computer systems, 1975), minimizing the risk of unintended side effects or privilege escalation.

Scheduling in POP is tightly coupled with pipeline-level concurrency guarantees (see §1.4). Worker functions themselves are stateless and are dispatched by the runtime into lightweight, elastic worker pools associated with each node. This model draws inspiration from actor-model scheduling (Agha, 1986) and dataflow execution strategies (Lee & Parks, Dataflow process networks, 1995), where units of computation are isolated and message-driven. By design, workers execute independently and do not share mutable state, thereby eliminating race conditions common in shared-memory concurrency models (Scott, 2016).

The lifecycle may be formalized as a sequence of stages:

1. **Dispatch**: A node receives an event and dispatches it to an available worker from its pool.
2. **Execute**: The worker function processes the event in isolation, optionally interacting with node-state tables or invoking helper functions.
3. **Emit or Error**: The worker emits a downstream Event<U> or signals failure by returning an Error<E>. Either output may be nil, ensuring that no unnecessary downstream traffic or error propagation occurs.
4. **Cleanup**: Any temporary resources (e.g., mutable scopes) are released, and the worker returns to the pool for reuse.

Elastic scheduling ensures that worker pools can scale up or down depending on system load, a property critical for balancing throughput and latency (Hellerstein, 2010). Importantly, because workers are defined as stateless by default, scaling decisions do not introduce nondeterministic side effects—a frequent challenge in traditional event-driven systems.

By binding lifecycle semantics and scheduling policies to compile-time constructs, POP reduces runtime uncertainty. While nondeterministic scheduling is unavoidable in distributed environments, compile-time analyzability of worker dispatch and capability restrictions ensures that concurrency remains tractable, predictable, and formally verifiable.

#### **Scoped Mutability**

Worker functions in POP are pure by default, and immutability is enforced across function boundaries to preserve analyzability, concurrency safety, and determinism. However, there are legitimate cases—such as aggregation, intermediate computation, or resource preparation—where temporary mutability is both practical and necessary. To accommodate these cases, POP introduces the mutable() operator, which grants narrowly scoped mutability within a worker function.

The semantics of mutable() are designed to mirror the discipline of functional languages that allow controlled mutation without abandoning referential transparency. For instance, Haskell employs monads to encapsulate side effects (Peyton Jones, 2003), while ML-like languages have long supported region-based mutable references constrained by lexical scope (Tofte & Talpin, 1997). In POP, mutable() serves a similar purpose: it allows developers to create temporary mutable bindings that are strictly limited in duration and visibility. The compiler enforces scope boundaries, ensures that mutations cannot leak into the event stream, and inserts automatic cleanup instructions at the end of the block.

This approach is motivated by both theory and practice. Early work in dataflow systems demonstrated that uncontrolled mutability introduces race conditions and undermines static scheduling guarantees (Dennis & Misunas, 1975; Lee & Parks, Dataflow process networks, 1995). Similarly, programming language research highlights the tension between purity and performance: while immutability simplifies reasoning, bounded mutability often improves efficiency and reduces overhead (Scott, 2016). POP’s mutable() operator acknowledges this trade-off by making mutability explicit and analyzable at compile time, avoiding the pitfalls of implicit shared state.

Importantly, scoped mutability in POP is **ephemeral**. Once a mutable block ends, its bindings are automatically frozen, preventing further modification. This ensures that worker outputs remain deterministic with respect to their inputs, even when temporary mutable state is employed internally. Moreover, because worker functions are stateless in the pipeline context, no mutable data persists beyond the lifetime of a single event’s execution.

By restricting mutability to explicitly declared, tightly controlled contexts, POP balances the need for performance and flexibility with its paradigm-level guarantees of concurrency safety and formal analyzability. This design continues the tradition of “safe imperative islands within declarative seas” (Hudak, 1989), ensuring that worker functions can be expressive without compromising the larger system’s correctness.

#### Node-State API Integration

Although worker functions in POP are designed to be stateless, certain pipeline operations require limited, structured state for correctness and efficiency. Examples include aggregating partial results, caching intermediate computations, or coordinating deduplication across concurrent workers. To accommodate these needs without undermining POP’s concurrency guarantees, nodes may expose node-state tables, accessible only through a restricted API: .set(), .get(), .update(), and .list().

This design aligns with long-standing research in safe state management within concurrent and distributed systems. Shared mutable memory has historically been a major source of race conditions and nondeterminism (Lamport, 1978; Herlihy & Shavit, The art of multiprocessor programming (Revised 1st ed.), 2012). POP mitigates these risks by enforcing strong isolation: node-state tables are ephemeral, scoped to a single node, and inaccessible across node boundaries. They exist only for the program’s runtime, ensuring that no state persists beyond execution. Furthermore, all access methods implicitly enforce fine-grained locking and atomicity, ensuring that concurrent workers operate safely even under high load.

Compile-time analyzability distinguishes node-state tables from conventional shared memory. The compiler declares state tables as structural components of the pipeline, making their existence and permissible operations explicit. This permits static verification that state use adheres to POP’s safety rules and prevents emergent dependencies that could compromise analyzability (Lee & Parks, Dataflow process networks, 1995). Runtime enforcement complements this by applying method-level guarantees: .set() and .update() operations will include compile-time–defined timeouts, preventing deadlocks or unbounded blocking (§1.5.6.3).

By constraining workers to a fixed API, POP ensures that node-state tables remain specialized coordination tools rather than general-purpose mutable memory. This reflects a “safe subset” philosophy, similar to the use of abstract data types in programming language design (Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985), where controlled interfaces enforce invariants. Workers cannot bypass these methods or expose raw state, making state use both analyzable and observable.

In effect, node-state tables provide the best of both worlds: the performance benefits of localized shared state and the safety of declarative, analyzable concurrency models. They enable patterns like windowed aggregation or limited coordination while preserving POP’s foundational guarantees of determinism, isolation, and concurrency safety.

#### **Concurrency Within Workers**

While POP ensures concurrency safety at the pipeline level by eliminating shared mutable state between nodes, worker functions themselves may exploit concurrency internally to improve responsiveness or throughput. This is particularly useful for tasks such as parallelizing computationally expensive subroutines, issuing multiple independent I/O requests, or pipelining stages of a local algorithm.

POP’s design imposes strict boundaries on such concurrency. Worker functions are executed atomically relative to the pipeline: a worker may spawn internal lightweight threads or asynchronous tasks, but all such activity must resolve before the worker returns an output Event<U> or Error<E>. This ensures that the observable pipeline semantics remain deterministic with respect to the event being processed, regardless of the concurrency model employed internally.

The restriction against shared mutable state across threads extends into the worker scope. Internal tasks cannot rely on unsynchronized global variables or external side effects. Instead, concurrency is structured around immutable data sharing, message-passing, or the use of the node-state API (§1.5.6.3), which provides safe, compiler-verified access to ephemeral shared state. By disallowing implicit concurrency hazards, POP avoids nondeterministic outcomes common in thread-based systems (Herlihy & Shavit, The art of multiprocessor programming (Revised 1st ed.), 2012; Scott, 2016).

This model reflects lessons from multiple paradigms. The actor model (Agha, 1986) demonstrated that concurrency can be structured around isolated processes communicating via messages, while Go’s goroutines and channels (Pike, 2012) showed that lightweight scheduling combined with explicit synchronization primitives yields practical scalability. POP draws from these traditions but adapts them to its declarative foundations: concurrency within a worker is permitted, but bounded by the requirement that all local tasks complete before downstream propagation.

In effect, concurrency inside workers provides developers with local flexibility without sacrificing global analyzability. Complex computations can exploit parallel resources, but the pipeline as a whole remains analyzable, reproducible, and free of emergent synchronization bugs. This balance ensures that POP can scale to modern workloads while maintaining the strict safety guarantees that distinguish it from conventional event-driven systems.

#### Internal Worker Concurrency and Pipeline Determinism

Concurrency inside worker functions introduces opportunities for local performance optimization, but it must not compromise the determinism of the pipeline as a whole. POP enforces this by requiring all asynchronous tasks spawned within a worker to resolve before the worker emits either an Event<U> or an Error<E>. This ensures that from the perspective of the pipeline, each worker execution remains atomic, preserving analyzability and reproducibility (Lee & Parks, Dataflow process networks, 1995).

This balance reflects lessons from actor-model scheduling (Agha, 1986) and languages such as Go, where lightweight concurrency primitives are isolated within well-defined scopes (Pike, 2012). POP refines this further by embedding concurrency into a declarative context: concurrency is allowed locally, but determinism is enforced globally. In this way, POP eliminates emergent nondeterminism that has historically plagued event-driven and shared-memory systems (Herlihy & Shavit, The art of multiprocessor programming (Revised 1st ed.), 2012).

#### Node API Access (Ingress()/Egress() Boundaries)

Ingress() and Egress() nodes serve as the only sanctioned boundaries for external I/O in POP. Worker functions within these nodes may interact with files, devices, networks, or system resources, but only through capabilities declared at compile time (Levy, 1984; Saltzer & Schroeder, The protection of information in computer systems, 1975). For example, an ingress worker may be authorized for FileRead:/opt/\*.json, while an egress worker may be authorized for NetOut:api.example.com:443.

This restriction reduces the attack surface and enforces analyzability: all possible I/O operations are visible in the pipeline’s declarative structure. It also isolates failures, ensuring that I/O errors are captured and rerouted via the error pipeline (see §1.1.8). In effect, ingress and egress APIs embody POP’s principle of least privilege while maintaining operational flexibility.

#### Node API Access (Transform() Boundaries)

Transform() nodes represent the core computational units in POP. Worker functions within Transform() nodes cannot perform external I/O; instead, they operate on immutable event payloads and may produce new events or errors. Their only external interactions are through helper functions and node-state tables (§1.5.6.3).

This design enforces separation of concerns: computation is isolated from I/O, allowing Transform() workers to be pure and analyzable. It mirrors principles of functional programming, where functions are side-effect free and results depend solely on inputs (Hudak, 1989; Scott, 2016). By disallowing implicit side effects, POP ensures that Transform() nodes remain reliable building blocks for both optimization and formal verification.

#### Node API Access (FanOut() and Collect())

FanOut() and Collect() nodes differ from Ingress(), Egress(), and Transform() in that their workers are compile-time–generated artifacts rather than programmer-defined functions. Their behavior is determined entirely by the pipeline declaration:

* FanOut()—workers route events across multiple downstream paths according to static rules or type constraints.
* Collect()–workers aggregate events from multiple upstream edges into a single flow.

Because these nodes are declarative in nature, their semantics are highly optimizable. The compiler can reason about their routing and merging logic at compile time, enabling advanced scheduling and correctness guarantees (Dennis & Misunas, 1975). This contrasts with the programmer-defined logic in Ingress(), Transform(), or Egress(), which must be sandboxed and analyzed separately.

Together, these API boundaries provide a clean separation between declarative and imperative concerns. Ingress() and Egress() connect pipelines to the outside world, Transform() nodes provide programmer-defined computation, and FanOut()/Collect() enable structural composition. By making these distinctions explicit, POP reduces semantic drift and ensures that pipelines remain analyzable, predictable, and secure.

#### Execution Context Summary

The execution context of worker functions defines the precise boundary between imperative computation and declarative pipeline structure in POP. By formalizing lifecycle and scheduling rules (§1.5.6.1), enforcing scoped mutability (§1.5.6.2), and introducing node-state APIs for controlled coordination (§1.5.6.3), POP ensures that local flexibility does not compromise global determinism. Internal concurrency is permitted (§1.5.6.4–1.5.6.5), but always bounded by the requirement that workers complete atomically with respect to their input events, preserving analyzability and reproducibility at the pipeline level.

Furthermore, API distinctions among node types (§1.5.6.6–8) reinforce separation of concerns: Ingress() and Egress() define explicit I/O boundaries, Transform() nodes isolate computation, and FanOut()/Collect() remain purely declarative structures. This layered model yields a programming paradigm in which imperative logic is both powerful and disciplined, providing developers with expressive tools while ensuring that safety, concurrency, and observability remain enforceable at compile time.

## Observability and Telemetry

Figure 11: FanOut to Logs Example

A diagram of a fanout

AI-generated content may be incorrect.

Observability in Pipeline-Oriented Programming (POP) is a first-class paradigm feature, not a secondary operational concern. While conventional event-driven and object-oriented systems often treat logging, metrics, and tracing as afterthoughts bolted onto existing architectures, POP integrates observability directly into the semantics of events, errors, and pipeline execution. This integration reflects the modern systems engineering perspective that reliability depends on the ability to measure and understand internal state from external outputs (Burns & Oppenheimer, 2019; Majors, In Praise of “Normal” Engineers , 2025). Every Event<T> and Error<E> object in POP carries structured metadata designed for introspection, enabling developers and operators to trace flows, diagnose bottlenecks, and reason about correctness both at compile time and in production. In this way, observability and analyzability reinforce one another, ensuring that POP programs can be validated not just in theory but in practice.

### Principles of Observability in POP

POP adopts observability as a design principle rooted in three complementary guarantees:

1. **Events as observable units**: Every event carries a globally unique EventId (see §1.1.5.3) and embedded metadata, ensuring that its provenance, path, and transformations are always reconstructable.
2. **Error transparency**: Errors are never hidden or silently discarded. Instead, every failure is wrapped in an Error<E> object with contextual metadata and routed through error pipelines (§1.1.8), making error handling itself observable.
3. **Declarative visibility**: Because pipelines are declared as static graphs, all observability points—Ingress(), Transform(), FanOut(), Collect(), and Egress()—are explicit and analyzable at compile time.

This approach distinguishes observability in POP from simple logging. Logs often capture partial, ad hoc signals of program behavior; observability, by contrast, emphasizes the ability to ask arbitrary questions of a system without prior knowledge of the failure modes (Majors, Observability engineering: Achieving production excellence., 2018). By embedding observability at the level of language constructs rather than relying on developer-added log statements, POP guarantees a uniform, analyzable telemetry fabric across all pipelines.

The principles also align with established best practices in distributed systems. Google’s Site Reliability Engineering (SRE) model stresses that monitoring must capture **“**goldensignals**”** such as latency, traffic, errors, and saturation in order to make systems operable at scale (Burns & Oppenheimer, 2019). POP enforces similar visibility through event metadata, backpressure signals, and error pipelines, but does so as part of the programming model itself. This integration ensures that POP applications inherently satisfy operational observability requirements, reducing the cognitive load on developers and operators alike.

### Event-Level Telemetry

At the most granular layer, observability in POP begins with the event itself. Each Event<T> is not only a carrier of application data but also an observable unit, enriched with metadata that documents its origin, processing path, and execution context. This design ensures that every event can be uniquely identified, traced through the pipeline, and correlated with system performance.

A unique EventId is generated at ingress, composed of the ingress node identity, worker identifier, and a 64-bit sequential number (see §1.1.5.3). This identifier is guaranteed to be unique within the lifetime of a program execution, providing sufficient granularity for production analysis while avoiding the overhead of global uniqueness. Event metadata additionally records timestamps, node identifiers, and worker pool identifiers, enabling operators to reconstruct event lifecycles from Ingress() to Egress().

Errors are treated with equal transparency. Rather than discarding failures or surfacing them only through logs, POP requires that each failure be encapsulated as an Error<E>, which itself contains the related Event<T> and error-specific metadata. This approach ensures that error handling is fully observable, and that debugging can be tied directly to the events that triggered failure conditions. Similar strategies have proven invaluable in distributed stream processing frameworks such as Apache Flink, where event-time and error-time metadata enable fault tolerance and recovery (Carbone, et al., 2015).

The metadata model extends beyond fault isolation: it provides the foundation for runtime telemetry. By embedding observability data into events themselves, POP enables pipelines to measure throughput, latency, and resource utilization without requiring additional logging or side-channel instrumentation. This principle mirrors the “golden signals” of latency, traffic, errors, and saturation emphasized in SRE practice (Burns & Oppenheimer, 2019). In POP, those signals are inherently captured by the event model, ensuring that telemetry is consistent, complete, and analyzable at compile time.

Thus, event-level telemetry transforms events from opaque data carriers into rich observability artifacts. By requiring identifiers, metadata, and structured error encapsulation, POP guarantees that every event flowing through a pipeline can be observed, analyzed, and reasoned about—both in real time and retrospectively.

### Pipeline-Level Telemetry

While event-level observability ensures that each Event<T> is traceable and analyzable, pipeline-level telemetry provides the broader operational perspective required to manage performance, diagnose systemic issues, and optimize throughput. In POP, pipelines are not opaque sequences of functions but explicitly declared graphs, and this structure enables telemetry to be both analyzable at compile time and enforceable at runtime.

At this level, the system captures aggregate measures such as latency distributions, throughput, queue depth, and backpressure events across nodes. These metrics allow developers and operators to assess not only how individual events are processed but also how the pipeline as a whole behaves under varying workloads. The importance of such end-to-end visibility has long been recognized in distributed systems research (Crosby & Wallach, 2003; Dean & Barroso, The tail at scale, 2013), where tail latencies and uneven load distribution can disproportionately impact user-facing performance.

POP integrates these concerns directly into its telemetry model. Each edge in the pipeline graph inherently records its scheduling, buffering, and backpressure behavior, ensuring that delivery semantics (§1.1.7.3) and scheduling guarantees (§1.4.6) are observable in practice. By making these properties measurable, POP avoids the common pitfall of emergent, undocumented behaviors that plague traditional event-driven frameworks.

Furthermore, pipeline-level telemetry enables capacity planning and elasticconcurrency (§1.4.4.3). Because ingress rates, worker pool utilization, and fan-out/collect node behaviors are observable, operators can tune concurrency strategies or introduce new resources in response to measurable saturation rather than reactive guesswork. This aligns closely with the “golden signals” emphasized in site reliability engineering: latency, traffic, errors, and saturation (Burns & Oppenheimer, 2019)

Finally, pipeline-level telemetry ensures that error handling pipelines (§1.1.8) are not black-box mechanisms. By measuring error rates, error classifications, and their propagation through dedicated error pipelines, POP guarantees that fault-handling is itself observable and subject to the same analytical rigor as normal event flow.

By combining event-level and pipeline-level telemetry, POP provides a two-tier observability model: one focused on fine-grained event identity and another on systemic performance and resilience. This layered approach ensures that operators can move seamlessly between micro-level debugging and macro-level capacity management.

### Node and Worker-Level Metrics

While pipeline-level telemetry (§1.6.3) provides a systemic perspective, effective observability also depends on **fine-grained node and worker metrics.** In Pipeline-Oriented Programming (POP), nodes are the fundamental processing units, and workers are the imperative execution contexts that carry out the declared transformations, ingress, or egress operations. Monitoring at this level ensures that performance anomalies, resource contention, and localized failures can be detected and mitigated before they propagate through the broader system.

#### Node Metrics

Each node exposes telemetry on its event processing rate, queue depth, latency distribution, and error frequency. For example, a Transform() node reports not only how many events it processes per unit time but also the distribution of worker execution durations. Such data aligns with well-established practices in distributed streaming frameworks (Carbone, et al., 2015; Kreps, Narkhede, & Rao, Kafka: A Distributed Messaging System for Log Processing., 2011), where bottlenecks at individual operators often dominate system performance.

Additionally, nodes report backpressure signals (§1.1.7.3), making delivery semantics analyzable in practice. If a node consistently applies drop policies (e.g., dropOldest), telemetry reveals this degradation explicitly rather than leaving it implicit. This ensures that performance regressions are observable at the exact locus where they occur.

#### Worker Metrics

Worker-level metrics provide insight into the execution health of imperative functions within nodes. These include:

* CPU and memory utilization of worker pools.
* Error rate and classification (e.g., WorkerRuntimeError, WorkerCapacityReached).
* Scheduling delays, showing how long events wait before being dispatched to a worker.
* Sandbox violations or denied capability attempts, which are logged to expose security anomalies.

Such metrics echo the principle of “resource isolation with visibility” emphasized in operating system and concurrency research (Agha, 1986; Dean & Barroso, The tail at scale, 2013), ensuring that even within shared infrastructure, the cost and behavior of individual workers are measurable.

#### **Linking Node and Worker Telemetry**

The true strength of POP’s observability model lies in the **integration of node and worker metrics into the event tracing model (§1.6.1).** Each Event<T> carries its EventId, enabling developers and operators to correlate specific event latencies or failures with the exact worker instance and node path. This provides both **vertical observability** (tracing an event through the pipeline) and **horizontal observability** (understanding node-level saturation or worker-level failures).

By binding node- and worker-level telemetry to the declarative structure of the pipeline, POP ensures that debugging and optimization remain analyzable and predictable. Rather than depending on ad hoc logging or external instrumentation, the paradigm embeds observability directly into the language-level abstractions.

### **Error and Anomaly Telemetry**

Errors in distributed and concurrent systems are inevitable, but how they are surfaced and acted upon determines system resilience and developer productivity. In POP, error handling (§1.1.8) is a **first-class language construct**, and telemetry plays a crucial role in ensuring that errors are not only captured but also made analyzable at runtime.

#### **Structured Error Reporting**

All errors are encapsulated in Error<E> objects, which preserve the context of the originating Event<T> alongside error-specific metadata. This structure allows telemetry systems to classify errors by type (e.g., ParseError, ValidationError, WorkerRuntimeError) and to correlate them with the affected event, node, and worker instance. Such structured error representation aligns with research emphasizing the value of type-safe error handling and traceability in concurrent systems (Agha, 1986; Schneider, 2000).

#### **Error Pipeline Telemetry**

Errors are routed through dedicated **error pipelines** (§1.1.8), which are declaratively analyzable structures rather than ad hoc runtime handlers. Telemetry hooks into these pipelines provide:

* **Error rates by type and source** (e.g., frequency of validation failures in a Transform() node).
* **Propagation analysis**, showing whether an error was successfully handled or escalated to the global ErrorPipeline().
* **Retry statistics**, including counts of successful retries and terminal failures.

This structured error telemetry ensures that no error path is silent, reducing the likelihood of “dark failures” where issues go undetected (Dean & Barroso, The tail at scale, 2013).

#### **Anomaly Detection and Policy Violations**

Beyond explicit errors, POP surfaces anomalies arising from **policy violations or unexpected runtime behavior**. Examples include:

* **Backpressure violations** (e.g., repeated worker pool saturation despite configured concurrency).
* **Capability violations** (e.g., a worker attempting unauthorized I/O).
* **Scheduling anomalies** (e.g., excessive event queuing delays suggesting imbalance).

These anomalies are logged and traced with the same rigor as errors, reinforcing observability across both correctness and performance dimensions. Such practices reflect modern approaches in reliability engineering, where anomaly telemetry complements error reporting to provide holistic visibility (O’Reilly Media, 2016; Google SRE, Beyer et al., 2016).

#### **Integration with Event Tracing**

Error and anomaly telemetry are not isolated systems but integrated with the **event tracing model (§1.6.1)**. Each error carries the originating EventId, making it possible to reconstruct causal chains across both normal and exceptional execution paths. This allows developers to trace how a single event failure propagates through a pipeline, enhancing diagnosability and aligning with end-to-end observability principles (Hellerstein, 2010).

### Longitudinal Analysis and Forecasting

While real-time telemetry provides essential visibility into the current state of a POP system, **longitudinal analysis** extends this perspective by examining trends across time. By aggregating telemetry streams over days, weeks, or months, POP enables developers and operators to move beyond reactive monitoring toward **predictive insights and proactive capacity planning.**

#### **Trend Analysis and Capacity Planning**

Event throughput, latency distributions, error rates, and backpressure statistics can be analyzed longitudinally to identify recurring workload patterns (e.g., diurnal load cycles or seasonal spikes). These analyses inform capacity planning decisions, such as pre-scaling worker pools or reconfiguring ingress scheduling. Similar practices are emphasized in large-scale distributed systems, where longitudinal telemetry has been used to anticipate "tail latency" risks (Dean & Barroso, The tail at scale, 2013) and improve elasticity strategies.

#### **Anomaly Detection Across Time Horizons**

Short-term anomalies may appear transient, but when correlated longitudinally they can reveal systemic vulnerabilities. For example, repeated bursts of BackpressureEvent in a single node may indicate deeper issues in pipeline balancing. Research in reliability engineering highlights that historical baselines are essential for distinguishing signal from noise in anomaly detection (Beyer, Jones, Petoff, & Murphy, 2016). By comparing current error and latency patterns against historical telemetry, POP systems can detect **slow burn failures** that would otherwise remain invisible.

#### **Forecasting with Predictive Models**

POP’s structured telemetry lends itself to **predictive modeling and machine learning** approaches. For instance, forecasting models may predict when worker pool saturation is likely to occur given observed growth in ingress rates, or when error rates trend upward under specific conditions. Prior work in adaptive systems suggests that predictive approaches improve both performance efficiency and resilience by shifting from reactive to proactive operational policies (Gandhi, Harchol-Balter, Das, & Lefurgy, 2012).

#### **Implications for Program Design**

By embedding longitudinal observability into the POP paradigm, program design itself becomes informed by **historical insight**. Developers can refine pipeline structures, reconfigure backpressure policies, or redesign worker functions not only based on immediate correctness but also on long-term operational behaviors. This closes the loop between **design-time analyzability** and **runtime** evidence, reinforcing POP’s goal of unifying declarative guarantees with operational safety.

### **Integrating Telemetry with Development and Operations**

Observability in Pipeline-Oriented Programming (POP) is not a passive byproduct but a first-class design principle. Telemetry collected at the event, node, and pipeline levels (see **§**1.6.1–1.6.6) becomes most effective when it is integrated directly into **development** and **operations** practices. This integration ensures that the guarantees of analyzability and concurrency safety extend beyond compile-time verification into runtime monitoring, debugging, and long-term system evolution.

#### **Development Feedback Loops**

Telemetry enables developers to close the gap between **specification and execution**. Metrics and traces expose how declarative pipelines behave under real workloads, validating assumptions about throughput, backpressure, and event delivery semantics. This reflects the principle articulated in continuous delivery research: rapid feedback from production is essential for improving software quality (Humble & Farley, 2010). In POP, observability feeds directly into the refinement of worker functions and pipeline graphs, reducing semantic drift between design-time models and runtime realities.

#### **Operational Practices and SRE Alignment**

For operators, telemetry supports the core Site Reliability Engineering (SRE) practices of **monitoring, alerting, and capacity planning** (Beyer, Jones, Petoff, & Murphy, 2016). Structured event metadata, combined with node-level and pipeline-level metrics, maps naturally onto service-level indicators (SLIs) and service-level objectives (SLOs). For example, latency distributions and backpressure metrics can be tied directly to user-facing SLOs for responsiveness, while error pipeline metrics inform system reliability goals. POP’s insistence on structured, analyzable telemetry reduces the ad hoc instrumentation burden that often hampers observability in event-driven systems.

#### **Bridging Development and Operations**

A distinctive strength of POP is how telemetry bridges development and operations by design. Developers gain visibility into runtime behavior through structured traces, while operators can rely on the same data for incident response and capacity management. This **shared observability fabric** embodies the DevOps principle of collapsing silos (Kim, Humble, Debois, & Willis, 2016), ensuring that both roles reason from a common operational truth.

#### **Toward Self-Adaptive Pipelines**

The integration of telemetry also lays the foundation for **self-adaptive pipelines**. By combining real-time metrics with historical forecasting (**§1.6.6**), systems may automatically tune worker pool sizes, backpressure policies, or error-handling strategies. Research in autonomic computing emphasizes such closed feedback loops as essential for managing complexity at scale (Kephart & Chess, 2003). POP’s structured approach to event semantics and error pipelines provides a natural substrate for these adaptive mechanisms.

## **Event Typing and Schema Contracts**

Event typing in Pipeline-Oriented Programming (POP) provides the foundation for analyzability, safety, and interoperability. Since pipelines consist of Event<T> objects flowing between nodes, type safety and schema contracts govern the semantics of data movement, transformation, and validation. This section outlines how POP enforces typing, manages schema evolution, and maintains compositional integrity across packages.

### **Strong Typing of Events**

Pipeline-Oriented Programming (POP) adopts a strong static typing discipline as a foundation for program safety, analyzability, and correctness. All data objects are statically typed, and every function—whether a worker or a helper—requires statically defined inputs and returns. At the pipeline level, this principle manifests in the fact that all events are represented as Event<T> objects, where T denotes the precise, compile-time–defined type carried by the event. Similarly, all error events are expressed as Error<E>, with the error wrapping and preserving the associated Event<T> (represented as E) type for traceability and analysis[[4]](#footnote-4).

Edges, which connect nodes in a pipeline, are also statically typed. An edge that accepts an Event<T> cannot accept an Event<U>, where U ≠ T. This invariant is compiler-enforced, ensuring that only type-consistent data flows across the graph. By embedding type safety into the structure of both events and their interconnections, POP eliminates entire classes of runtime errors related to type mismatches.

The advantages of this strong typing discipline are twofold. First, it enhances program safet**y** by guaranteeing that incompatible data cannot cross pipeline boundaries, even in complex composed systems. Second, it enables program analyzability, as type information provides a static foundation for reasoning about pipeline behavior, optimization opportunities, and security properties. These advantages echo longstanding arguments in programming languages research, where type systems are recognized as mechanisms for both correctness and expressiveness (Pierce, 2002; Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985).

By elevating strong typing from individual functions to the entire pipeline graph, POP ensures that type correctness is not merely a language feature but a paradigm-level guarantee. This makes POP pipelines predictable, formally analyzable, and safe for composition across diverse packages and runtime environments.

### Schema Evolution and Compatibility

While strong typing ensures that events are consistent and analyzable at compile time, long-lived systems require schema evolution to adapt to changing requirements. In Pipeline-Oriented Programming (POP), event schemas are static compile-time artifacts. Nonetheless, they must evolve in a manner that preserves both type safety and operational continuity between builds. This tension between rigidity and flexibility is a recurring theme in programming language and database research (Fowler, 2012; Apache Avro 1.7.7 Specification, 2014).

By expressing pipelines in terms of a language, which is compiled from source to create a binary artifact, much of the schema management issues affecting large, complex systems is eliminated at compile time. What remains is isolated to program interconnections at Ingress() or Egress()see (**§1.7.3**). Any pipeline (P) which consumes some reusable segment (S) must have a strongly typed contract, expressed as a type. If P and S both agree to accept Event<T> as their format of data exchange, then S is updated to accept Event<U>, the program defining P will break at compile time. Either P must be updated to accept Event<U> or S must be consumed as its earlier version. This creates a problem where POP must version packages leading up to compile time, and this is consistent with prior research (Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985; Pierce, 2002).

In POP, the package must have an independent semantic version. This version must be used when the main package imports S for use with the pipeline P. The package import must ensure through some rule that the version of S imported is compatible with pipeline P to avoid compile-time issues. This aligns with long-standing guidance on module systems and avoids “diamond dependency” ambiguity (Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985; Kleppmann, 2017).

When a consuming package (P) consumes a modular package (M), P imports M with a declarative rule:

Figure 12: Package Import versioning example

|  |
| --- |
| import m == v0.0.1 |

Or, alternatively,

Figure 13: Complex Boolean Package Import Rules

|  |
| --- |
| import m >= v0.0.1 && != v0.0.3 |

When P is compiled, it is satisfied with the import if the version of package M is v0.0.1 in the first case or if it is this version or newer in the second. This is the first check when sources are imported. POP supports backward, forward and full compatibility using powerful Boolean expressions, allowing a programmer to implement simple import rules or more complex constructs[[5]](#footnote-5). The result of this strategy is an inter-package contract settled at compile time, implementing practices consistent with the literature (Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985; Lee & Parks, Dataflow process networks, 1995; Pike, 2012; Parnas, 1972).

### Contracts Between Programs

What remains of the versioning issue exists at the Ingress() and Egress() nodes. POP has no opinion or standard for how this should be addressed. This could be solved by linguistic construct, or it may be addressed by program design. Future versions of POP may consider novel binary formats and cryptographic means of establishing clear, versioned contracts between POP-compatible programs. Such contracts will likely function much like interface definition in traditional programming languages, but at the higher abstraction of event flow and pipeline composition (Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985; Pierce, 2002). This will also likely involve standards for interconnected programs across the host boundary. The reader is encouraged to keep these points in mind when implementing any POP language.

### Type Safety vs. Flexibility Trade-Offs

Strong static typing and compiler-enforced schema contracts in POP provide significant benefits: they enhance program analyzability, prevent many classes of runtime errors, and ensure that cross-package pipelines remain verifiable at compile time. These guarantees draw upon long-standing results in programming language theory that demonstrate how strong typing improves safety and reasoning about program correctness (Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985; Pierce, 2002).

Yet these same guarantees impose trade-offs. Strict typing and schema enforcement can limit rapid prototyping, cross-version interoperability, and integration with heterogeneous external systems. Languages and frameworks that emphasize dynamic flexibility—such as JavaScript with duck typing or schema-less message-passing protocols—offer faster iteration but at the cost of reduced guarantees and more fragile runtime behavior (Abadi, Cardelli, Pierce, & Plotkin, 1991). POP chooses to privilege safety and analyzability, but it also recognizes the need for adaptability. This is why contracts allow controlled schema evolution (as defined in 1.7.2) and why helper functions can mediate between pipelines with different but compatible types.

The challenge is to balance rigidity and openness. Too rigid a system risks obsolescence as schemas evolve and external systems change; too flexible a system undermines the analyzability and safety guarantees that make POP distinctive. By making contracts explicit, requiring fully-qualified types across packages, and embedding schema versioning into the compilation process, POP mitigates this tension while acknowledging that type systems must evolve with practice.

In sum, POP deliberately accepts some constraints on flexibility to deliver stronger correctness guarantees. These constraints are not accidental limitations but principled design choices, informed by established theory in type systems and by the practical realities of building distributed, event-driven pipelines.

## Program Structure and Composition

Pipeline-Oriented Programming (POP) applications are not monolithic; they are constructed from modular units that combine declarative pipelines with imperative worker functions. Each program integrates packages, namespaces, entry points, error pipelines, and deployment artifacts into a cohesive whole. By codifying these structures at compile time, POP ensures that program organization is analyzable, composable, and predictable, while still allowing flexibility in runtime execution. This layered design aligns with long-standing principles of modular programming (Parnas, 1972) and modern distributed systems engineering (Hennessy & Patterson, 2019), offering developers both clarity of structure and safety of execution.

### Package and Namespace Organization

At the foundation of POP program structure lies the package and namespace system. Packages serve as the organizational boundary for related pipeline segments, worker functions, and type definitions. By grouping semantically related components together, packages facilitate modular development, code reuse, and versioned distribution across projects. Namespaces, in turn, provide a mechanism to prevent symbol collisions and ensure unambiguous references when multiple packages are composed into a single program.

In POP, a pipeline segment must be defined within a single package namespace. This rule ensures that all node definitions (Ingress(), Transform(), FanOut(), Collect(), and Egress()) and their associated worker functions are scoped consistently, avoiding the ambiguities that arise from ad hoc cross-module references. Cross-package composition is supported but requires fully qualified names, making dependencies explicit and analyzable at compile time. For example, a pipeline segment defined in analytics.telemetry may invoke another in core.ingestion, but the compiler ensures that type contracts and error-handling semantics are validated before linking.

This model draws on well-established principles from modular and component-based software engineering. As Parnas (1972) argued, information hiding and module boundaries are essential for program maintainability and evolvability. Similarly, namespace systems in modern languages such as C++, Java, and Go demonstrate the practical importance of avoiding symbol ambiguity and promoting separation of concerns (Stroustrup, 2003; Scott, 2016). By enforcing strict namespace rules, POP provides not only organizational clarity but also analyzability: pipelines can be reasoned about as independent modules that compose safely into larger systems.

In summary, the package and namespace system in POP provides:

* **Encapsulation** — Each pipeline segment is defined within a single package boundary.
* **Explicit composition** — Cross-package invocations require fully qualified names and compiler validation.
* **Safety** — Type contracts and error semantics are enforced across namespace boundaries.
* **Modularity** — Programs can evolve as distributed collections of reusable packages.

Through these rules, POP ensures that program structure is both modular and verifiable, laying the groundwork for analyzable pipeline composition in subsequent sections.

### Entry Points and Pipelines

In Pipeline-Oriented Programming (POP), program execution begins at Ingress() nodes, which define the entry points through which external inputs are admitted into the declarative pipeline graph. Because I/O is strictly confined to Ingress() and Egress() boundaries, these entry points represent not only the functional start of computation but also the security and trust boundaries of the program (Levy, 1984; Saltzer & Schroeder, The protection of information in computer systems, 1975). Each ingress node is associated with a worker function responsible for transforming external inputs into typed Event<T> objects, which then flow through the pipeline according to declarative edges.

Pipeline composition in POP extends beyond single entry points. A full program may consist of multiple pipeline segments, each rooted in an ingress node and potentially converging or diverging through intermediate transforms, fan-outs, and collections. Segments can be composed across namespaces and packages, provided their type signatures align and their capabilities are compatible. This modularity ensures that small, independently testable pipelines can be assembled into larger, production-scale programs without semantic drift (Hennessy & Patterson, 2019).

A distinguishing feature of POP is that **entry points are not inherently imperative main routines** as in procedural programming. Instead, they are declarative constructs instantiated by the compiler into load-balanced worker fleets, each capable of concurrent event processing. This design allows developers to scale pipelines elastically while preserving the analyzability and safety guarantees of declarative specifications (Agha, 1986; Lee & Parks, Dataflow process networks, 1995).

By treating ingress nodes as first-class program entry points, POP offers a structural model where pipelines compose deterministically and predictably. The explicit representation of entry points as part of the declarative program graph makes the overall structure analyzable at compile time, while the constrained imperative shell of worker functions ensures runtime flexibility where needed. Thus, entry points and pipelines in POP act as the architectural scaffolding that binds declarative correctness with operational pragmatism.

### Error Pipelines as Structural Backstops

In POP, error handling is not an afterthought or runtime patch but a first-class, compile-time construct. Every program includes a built-in ErrorPipeline(), a globally accessible structural component into which all Error<E> objects are routed when events fail validation, encounter runtime exceptions, or violate delivery guarantees. By embedding error handling directly into the program structure, POP ensures that recovery, visibility, and traceability are consistently available across all pipelines.

By default, ErrorPipeline() terminates at an Egress() node wired to standard error output, causing program termination when unhandled errors propagate. However, developers can reconfigure this pipeline declaratively, replacing the default egress with more sophisticated handlers—for example, redirecting errors to persistent logs, distributed queues, or automated retry sub-pipelines. Because ErrorPipeline() is defined declaratively, it benefits from the same static analyzability, type safety, and concurrency guarantees as functional pipelines, while remaining flexible enough to support operational customization.

The placement of a global error pipeline also reinforces POP’s emphasis on **observability and safety**. All errors are represented as Error<Event<T>> objects, preserving the metadata and EventId of the originating event. This design choice guarantees that error handling pipelines do not operate on decontextualized failures but always retain a traceable link to the original execution path. Such traceability supports both debugging in development and compliance requirements in production (Chen, Kiciman, Fratkin, Fox, & Brewer, 2002; Ousterhout, 2018).

Error pipelines in POP act as **structural backstops**, ensuring that all failures are captured, typed, and explicitly handled. This approach mirrors practices in reliable distributed systems where dedicated error channels reduce hidden failure modes and improve fault containment (Saltzer & Schroeder, The protection of information in computer systems, 1975; Dean & Barroso, The tail at scale, 2013). By elevating error handling to a declarative pipeline structure, POP formalizes resilience as an architectural guarantee rather than a runtime convention.

### Linking and Deployment

The final stage in the POP programming lifecycle is linking and deployment, where declarative pipeline graphs, worker functions, and supporting packages are transformed into an executable program. This process mirrors the traditional compilation and linking phases in systems programming but is specialized for pipeline semantics and concurrency guarantees.

At compile time, the POP compiler translates declarative pipeline definitions into intermediate representations that capture the structure of nodes, edges, and capabilities. Worker functions, written in a restricted imperative form, are compiled into sandboxed execution units (lightweight virtual machines) with only the resources permitted by their declared capabilities. The compiler then performs linking, which resolves package namespaces, type contracts, and pipeline segment invocations across packages. The result is a unified **program binary** embedding both the declarative pipeline graph and the imperative worker artifacts, enabling analyzability at runtime while preserving strong static guarantees.

Deployment extends this lifecycle into the operational domain. Once compiled, a POP binary encodes its pipelines as structural graphs that can be inspected, versioned, and verified before execution. The deployment runtime provisions worker pools, allocates resources, and configures ingress and egress nodes to interface with external systems. Because error pipelines are embedded alongside functional pipelines, fault tolerance is structurally enforced from the moment the program begins execution.

By integrating linking and deployment directly into the paradigm, POP ensures that program structure is analyzable before execution, reducing runtime uncertainty. This approach parallels the emphasis on whole-program reasoning in modern compilers ( (Cooper & Torczon, 2011; Muchnick, 1997) while adapting those principles to the declarative–imperative balance that defines pipeline programming. The result is a deployment process that is both predictable and flexible: predictable because type safety, capability enforcement, and pipeline structure are guaranteed at compile time; flexible because runtime elasticity allows worker pools to scale in response to workload demands.

## **Program-Level Semantics and Guarantees**

Once compiled and deployed, a Pipeline-Oriented Programming (POP) program offers a set of explicit, analyzable guarantees that distinguish it from ad hoc event-driven or message-passing systems. These guarantees are not universal solutions to all challenges of distributed computing—POP makes deliberate trade-offs to favor concurrency safety, analyzability, and operational transparency over maximum flexibility or expressiveness. By elevating safety, delivery, and observability to language-level constructs, POP ensures that programs can be reasoned about formally at compile time and monitored predictably at runtime.

### Safety Guarantees

POP enforces safety through compiler-verified invariants that extend across pipelines, nodes, and worker functions. At the foundation is **strong static typing**: all events are Event<T> objects, edges are type-bound, and all functions explicitly declare input and output types. This eliminates the possibility of passing ill-typed data across pipeline segments, a property aligned with the long-established benefits of strong typing in programming languages (Cardelli & Wegner, On Understanding Types, Data Abstraction, and Polymorphism, 1985; Pierce, 2002).

Complementing type safety is **immutability**. Events, once emitted, cannot be modified; instead, worker functions may transform them into new Event<U> objects. This model mirrors functional programming’s emphasis on immutability for concurrency safety (Peyton Jones, 2003) while retaining the operational clarity of dataflow models (Dennis & Misunas, 1975). Mutable state is only permitted in narrowly scoped contexts, such as the mutable() operator within a worker or node-state tables confined to node boundaries. These constructs are strictly controlled and analyzable, ensuring that concurrency hazards like race conditions remain detectable and bounded.

Finally, POP guarantees **concurrency safety** by disallowing shared mutable state between nodes and enforcing deterministic APIs for local state interactions. Worker functions execute in isolated, sandboxed contexts with no implicit I/O privileges, reducing the attack surface and minimizing nondeterministic behavior. This approach echoes Saltzer and Schroeder’s (1975) principle of least privilege, extending it into the concurrency domain: workers can do only what their declared capabilities and static types permit.

In aggregate, these safety guarantees make POP programs analyzable at compile time and predictable in distributed execution. Type safety prevents semantic drift between pipeline segments, immutability ensures event-level determinism, and concurrency safety eliminates the most common pitfalls of event-driven programming. The result is a paradigm in which correctness is not merely a property of careful programming practice but a structural guarantee of the language.

### Delivery Guarantees

POP defines delivery semantics as a function of **backpressure policy**, ensuring that guarantees are analyzable and predictable at compile time rather than emergent from runtime heuristics. This design avoids ambiguity common in event-driven or streaming frameworks, where delivery semantics often depend on runtime configuration or system load.

When a **block policy** is applied, upstream nodes suspend event emission until downstream capacity is restored. In this case, retries are permitted, and **at-least-once delivery** is the strongest achievable guarantee: every event is delivered at least once, though duplicates may occur in retry scenarios. This aligns with classic distributed systems findings that at-least-once semantics strike a balance between robustness and scalability (Skeen, 1983).

When **non-blocking policies** such as dropOldest, dropNewest, or shunt are in place, events may be discarded when downstream nodes cannot keep pace. Under these conditions, **at-most-once delivery** is the only enforceable guarantee: each event is delivered zero or one time, but never retried. This reflects practical trade-offs seen in streaming dataflow engines, where bounded queues and controlled loss improve latency and throughput at the cost of guaranteed delivery (Carbone, et al., 2015).

**Exactly-once delivery**, long a goal of distributed computing, is not a default guarantee in POP. It can only be approximated by programmer-defined worker logic, such as idempotent transformations or deduplication strategies. As Kreps (2014) and others have noted, exactly-once semantics are costly and often infeasible at scale; POP explicitly rejects this as a language-level promise, instead preferring analyzable guarantees bound to backpressure policy.

By making delivery semantics explicit in pipeline declarations, POP allows developers to reason formally about system behavior before deployment. Failures are not silent: if an event cannot be delivered or processed within its constraints, it is encapsulated as Error<Event<T>> and routed to the error pipeline, ensuring traceability and visibility. This integration of delivery semantics with backpressure, error handling, and observability creates a consistent model where performance, safety, and correctness reinforce each other rather than compete.

### Observability Guarantees

POP treats observability as a language-level guarantee rather than a runtime convenience. Every event in a pipeline is traceable by virtue of its EventId, a composite identifier formed from the ingress node identity, worker number, and a 64-bit sequential counter unique within a given program execution (see §1.1.5.3). This ensures that even when programs scale horizontally or replicate across workers, each event can be unambiguously tracked within the context of a program run.

In addition to EventId, all Event<T> and Error<E> objects carry **structured metadata** baked into the event model (see §1.1.6.3). Metadata includes timestamps, lineage information, and resource annotations, enabling developers and operators to correlate events across pipeline stages. Importantly, when errors occur, POP does not discard context: the error object is typed as Error<Event<T>>, encapsulating both the original event and error-specific metadata. This provides rich traceability for debugging, operational audits, and automated recovery pipelines.

At the **pipeline level**, observability is reinforced by built-in error pipelines and delivery semantics that ensure no failure is silent (see §1.6). Events that fail to process are explicitly routed into the error pipeline, where their metadata and EventId persist. This design echoes the Google SRE principle that reliability requires explicit instrumentation and visibility into system behavior (Beyer, Jones, Petoff, & Murphy, 2016), and Ousterhout’s (2018) emphasis that observability is essential to reasoning about software complexity.

By embedding traceability and metadata into the event abstraction itself, POP achieves **end-to-end observability guarantees**:

* Every event is identifiable via EventId.
* Every transformation preserves lineage through immutable wrapping of events.
* Every error is visible, typed, and traceable through the error pipeline.

These guarantees make POP programs analyzable during development and transparent in production, closing the gap between compile-time structure and runtime execution. Unlike ad hoc logging or tracing in traditional paradigms, POP enforces observability as part of its type system and pipeline semantics, ensuring that debugging, monitoring, and compliance are not optional add-ons but intrinsic properties of the model.

### Operational Guarantees and Limits

POP programs provide clear operational semantics at the language level, but these guarantees are intentionally scoped. The paradigm prioritizes analyzability, safety, and predictability, even at the expense of flexibility in some areas. By defining what is guaranteed—and equally, what is not—POP creates a transparent model for developers and operators.

First, POP guarantees **determinism at the structural level**. Pipelines are statically declared graphs with well-defined nodes, edges, and types. This allows the compiler to analyze flows, validate contracts, and enforce concurrency safety before deployment. However, **execution is not fully deterministic** at runtime: events may be scheduled in parallel across worker pools, and their precise interleaving is subject to the nondeterminism inherent in concurrent systems. This trade-off is deliberate, reflecting insights from dataflow research where bounded nondeterminism provides scalability without compromising analyzability (Lee & Parks, Dataflow process networks, 1995).

Second, POP guarantees **immutability and type safety** throughout execution, meaning events cannot be corrupted by shared mutable state. Yet it does **not guarantee exactly-once delivery**. As discussed in §1.9.2, delivery semantics are tied to backpressure policy: block policies ensure at-least-once delivery, while drop policies enforce at-most-once delivery. Exactly-once semantics are excluded because distributed systems research has consistently shown them to be costly and often infeasible at scale (Skeen, 1983; Kreps, Kafka: a distributed messaging system for log processing, 2014). POP instead emphasizes that failure cases are visible and traceable via error pipelines, ensuring that no event failure goes unobserved.

Third, POP enforces **observability guarantees** (see §1.9.3) but does not dictate operational policies such as log retention, storage replication, or alert thresholds. These remain system-level and organizational choices. POP ensures that the necessary metadata and error-handling structures are in place, but **operational practices** must be layered on top.

Finally, POP acknowledges **limits on flexibility**. Because pipelines are declared statically, certain dynamic behaviors—such as runtime graph mutation or unbounded recursion—are prohibited. These restrictions reduce expressiveness but allow the paradigm to enforce analyzability and safety guarantees that ad hoc event-driven systems cannot provide. As Dean and Barroso (2013) argue in the context of large-scale distributed systems, managing tail latency and complexity requires principled trade-offs rather than unbounded dynamism. POP’s design embodies this principle by balancing determinism with practical concurrency.

In summary, POP guarantees structural determinism, immutability, type safety, analyzable delivery semantics, and end-to-end observability. It does not guarantee exactly-once delivery, fully deterministic execution, or system-level operational outcomes such as durability or latency bounds. By being explicit about these boundaries, POP provides developers with a language model that is both predictable and transparent in its trade-offs.

### Comparative Perspective

Pipeline-Oriented Programming (POP) offers guarantees that position it differently from prior paradigms while drawing upon their foundational insights. Unlike Object-Oriented Programming (OOP), where program safety is dependent on encapsulation and disciplined use of mutable state) (Stroustrup, 2003; Scott, 2016), POP enforces immutability and type safety at the language level. This structural enforcement eliminates many concurrency hazards that OOP delegates to runtime mechanisms such as threads and locks.

Functional programming provides a closer analogy. Its emphasis on immutability and referential transparency (Peyton Jones, 2003) resonates with POP’s guarantees of event immutability and deterministic transformations. However, while functional languages often rely on higher-order functions and recursion to model complex flows, POP restricts expressiveness to five node types, trading flexibility for analyzability. This limitation reflects a deliberate design choice: to bind concurrency and safety guarantees directly to program structure.

Dataflow programming represents the closest antecedent to POP. As Lee and Parks (1995) showed, dataflow models provide analyzability and bounded nondeterminism, enabling static reasoning about scheduling and throughput. POP builds on this foundation but strengthens guarantees by coupling delivery semantics with backpressure policies and by embedding error pipelines into the language itself. Where classic dataflow leaves runtime policies underspecified, POP codifies them into declarative constructs.

The actor model offers another instructive contrast. Actors, as defined by Agha (1986), encapsulate state and communicate via asynchronous message passing. This model ensures isolation but provides no guarantees about delivery beyond eventual best effort. POP diverges by eliminating shared mutable state entirely and by defining delivery semantics at the pipeline level. Whereas actor systems rely on developer discipline and runtime libraries to enforce robustness, POP integrates safety, delivery, and observability guarantees directly into its type system and execution model.

Taken together, these comparisons underscore POP’s unique position. OOP excels at flexible modeling, functional programming at purity and immutability, dataflow at analyzability, and actors at isolation. POP synthesizes elements of each while sacrificing certain dimensions of generality to enforce strong guarantees. Its safety, delivery, and observability semantics are not emergent properties of runtime libraries but compiler-verified invariants, making it a paradigm designed for correctness at scale in concurrent and distributed systems.

### Summary

The first section of this work has articulated the theoretical underpinnings of Pipeline-Oriented Programming (POP), establishing it as a programming paradigm built on analyzability, immutability, concurrency safety, and structural guarantees. Where earlier paradigms such as object-oriented programming emphasized encapsulation (Stroustrup, 2003), functional programming purity (Peyton Jones, 2003), dataflow analyzability (Lee & Parks, Dataflow process networks, 1995), and the actor model’s isolation (Agha, 1986), POP synthesizes these insights into a coherent framework that binds program semantics to declarative pipeline graphs and strongly typed events.

POP begins by constraining program structure. Only five node types—Ingress(), Transform(), FanOut(), Collect(), and Egress()—define the flow of data, with each node operating on immutable, statically typed Event<T> objects. This minimalism is deliberate: it permits concurrency reasoning and delivery semantics to be determined at compile time, reducing emergent runtime nondeterminism. Backpressure policies define whether pipelines guarantee at-least-once or at-most-once delivery, making reliability properties explicit and analyzable rather than emergent.

Error handling is elevated to first-class status. Every POP program includes an ErrorPipeline(), a compile-time construct that ensures all failure modes are visible, analyzable, and recoverable. Events that fail processing are wrapped in Error<E> objects and routed through the error pipeline, preventing silent loss and enabling systematic handling of failure cases.

Security is addressed through capabilities, sandboxed workers, and strict ingress/egress boundaries. By limiting I/O to Ingress() and Egress() nodes, POP narrows the attack surface, enforces clear trust boundaries, and enables policy-driven runtime restrictions. Worker functions, while imperative in nature, are sandboxed in compiler-optimized VMs with only the minimal capabilities declared, preserving analyzability and limiting the impact of compromise.

Concurrency and scheduling are tightly integrated with the paradigm. Events flow asynchronously, but pipeline determinism is preserved by disallowing shared mutable state except through controlled mechanisms such as node-state tables. Scheduling policies balance determinism with runtime elasticity, leveraging lightweight worker processes to achieve scalable parallelism while retaining analyzability.

Worker functions themselves are deliberately constrained. Their signatures are uniform—accepting Event<T> and returning Event<U> or Error<E>—ensuring predictability across all nodes. While helper functions allow abstraction and reuse, worker functions must remain pure except where explicitly wrapped in the mutable() operator for controlled, local mutability. These restrictions embed safety and correctness directly into the programming model.

Finally, observability and telemetry are embedded at the language level. Every event carries metadata and a unique EventId, ensuring traceability across pipelines, while structured metrics and error pipelines provide longitudinal insight into system behavior. Together, these constructs ensure that POP programs are not only safe and analyzable, but also operationally observable.

In sum, Section 1.x has established POP as a paradigm that constrains expressiveness in favor of analyzability, determinism, and safety. By binding correctness, delivery guarantees, and observability into the structure of programs themselves, POP provides developers with a framework for constructing reliable concurrent and distributed systems where semantics are compiler-enforced rather than runtime emergent.

# POP Programming in AMI

AMI[[6]](#footnote-6) (the “Asynchronous Machine Interface”) is a Go-derived language that implements the Pipeline-Oriented Programming (POP) model specified in Chapter 1. AMI adds a declarative pipeline grammar, first-class struct and enum, capability-gated I/O, RAII memory management and analyzable delivery semantics. Pipelines are statically declared graphs; events are immutable values; and error handling is explicit via an error pipeline. Imperative work happens only inside sandboxed worker functions whose effects are bounded by capabilities (Saltzer & Schroeder, The protection of information in computer systems, 1975; Levy, 1984). These choices follow the dataflow lineage (Dennis & Misunas, 1975; Lee & Parks, Dataflow process networks, 1995), incorporate lessons from actor systems (Agha, 1986) while keeping a fixed graph, and embrace immutability improve reasoning and concurrency safety (Backus, 1978; Scott, 2016). AMI’s surface syntax remains Go-assimilating where practical (Donovan & Kernighan, 2015; Pike, 2012).

AMI consists of two sub-languages:

* **Declarative Pipeline Descriptor Language (DPDL):** declares a static pipeline graph (nodes, edges, capacities, backpressure, types).
* **Imperative Worker Function Language (IWFL):** defines node-local worker functions with a uniform, analyzable signature.

Throughout this chapter, we adopt the POP guarantees from Chapter 1.0 (immutability across node and function boundaries, explicit backpressure and delivery semantics, capability-bounded I/O, error pipelines, and worker-pool scheduling) and show how AMI realizes them in a concrete language.

## Lexical Structure

AMI inherits Go’s lexical conventions and extends them to support the POP model. Source text is processed as a stream of Unicode code points. Tokens are formed by the usual composition of identifiers, keywords, literals, operators, delimiters, and whitespace/comments, with no ambiguity introduced by POP’s declarative additions. Where AMI diverges from Go, it does so to preserve analyzability, immutability, and capability-bounded execution defined in Chapter 1.0.

### Source Text and Encoding

All AMI source files are UTF-8 encoded without a byte-order mark (BOM). Line endings are normalized to LF during lexical analysis; CR and CRLF are accepted as input but have no semantic effect. Each file belongs to exactly one package; file names and paths are outside the language proper and are handled by the toolchain.

### Identifiers

Identifiers denote names of packages, types, values, functions, methods, and pipeline nodes.

#### Identifier Pattern

Identifiers are case-sensitive and must match the pattern [A-Za-z\_][A-Za-z0-9\_]{0,1024}. This bound ensures predictable tooling and permits static analysis over large codebases without sacrificing readability. Examples include Ingress, cleanFunc, and db.ExecStoredProc.

#### Unicode and Identifier Policy

Identifiers are sequences of Unicode code points and must match [A-Za-z\_][A-Za-z0-9\_]{0,1024} for maximal tooling compatibility. For portability and analyzability, AMI compilers normalize identifier text to NFC and reject visually confusable mixed-script identifiers.

#### Exported Identifiers

Exported identifiers begin with an uppercase letter and unexported identifiers begin with a lowercase letter. Identifiers beginning with an underscore (\_) character are unexported.

### Keywords

AMI reserves Go’s keyword set and adds POP node constructors (ingress, egress, fanout, collect, transform), the error-pipeline constructor ErrorPipeline, and the scoped mutability operator mutable. The reserved word “go” and “chan”, however, are removed; AMI does not permit goroutine creation from user code, preserving analyzability and capability bounds.

Table 2 : AMI Keywords

|  |  |
| --- | --- |
| Keyword | Purpose |
| break | Exits the nearest for, switch, or select statement immediately. |
| case | Defines a branch in a switch or select statement. |
| const | Declares a compile-time constant. |
| continue | Skips the current iteration of a loop and proceeds to the next. |
| default | Specifies the fallback branch in switch or select. |
| defer | Schedules a function call to run after the surrounding function returns. |
| else | Defines the alternative branch in an if statement. |
| fallthrough | Forces execution to continue to the next case in a switch. |
| for | The only looping construct in Go; supports iteration and while-style. |
| func | Declares a function or method. |
| goto | Performs an unconditional jump to a labeled statement. |
| if | Defines a conditional execution branch. |
| import | Brings packages into the current source file. |
| interface | Declares an interface type, a set of method signatures. |
| label | Declares a target for a goto statement. |
| map | Declares a hash map (key-value store) type. |
| package | Declares the package for the current source file. |
| range | Iterates over elements in arrays, slices, maps, strings, or channels. |
| return | Returns from a function, optionally with values. |
| select | Waits on multiple channel operations. |
| struct | Declares a structured type with named fields. |
| switch | Multi-way branch statement based on values. |
| type | Declares a new type or type alias. |
| var | Declares a variable, optionally with initialization. |
| ingress | Represents a pipeline node constructor representing an entrypoint |
| egress | Represents a pipeline node constructor representing a pipeline exit/output node. |
| fanout | Represents a pipeline router/broadcast node which creates 1:many relationships. |
| collect | Represents a pipeline router/aggregation node which creates a many:1 relationship. |
| transform | Represents a pipeline data transformation node. |
| ErrorPipeline | Represents the default head for error handling pipelines. |
| mutable | Defines a mutable scope. |

### Pre-declared Identifiers and Reserved Words

AMI adopts Go’s predeclared types, constants, and built-ins (e.g., bool, int, string, len, make, nil). In addition, AMI predeclares the generic interfaces Event<T> and Error<E> to reflect POP’s first-class event and error objects and to make pipeline typing explicit at compile time (§1.1.6–§1.1.8). These names are always in scope and cannot be redefined.

Table 3: Reserved Words

|  |  |  |
| --- | --- | --- |
| Identifier | Category | Purpose / Meaning |
| bool | Type | Boolean type (true/false). |
| byte | Type | Alias for uint8, represents raw data. |
| complex64 | Type | Complex numbers with float32 real and imaginary parts. |
| complex128 | Type | Complex numbers with float64 parts. |
| error | Type | Built-in interface type for error handling. |
| float32 | Type | 32-bit IEEE-754 floating point. |
| float64 | Type | 64-bit IEEE-754 floating point. |
| int | Type | Signed integer, machine-dependent size (32 or 64 bits). |
| int8 | Type | 8-bit signed integer. |
| int16 | Type | 16-bit signed integer. |
| int32 | Type | 32-bit signed integer (rune is alias). |
| int64 | Type | 64-bit signed integer. |
| rune | Type | Alias for int32, represents a Unicode code point. |
| string | Type | Immutable sequence of bytes (UTF-8 by convention). |
| uint | Type | Unsigned integer, machine-dependent size. |
| uint8 | Type | 8-bit unsigned integer. |
| uint16 | Type | 16-bit unsigned integer. |
| uint32 | Type | 32-bit unsigned integer. |
| uint64 | Type | 64-bit unsigned integer. |
| uintptr | Type | Unsigned integer large enough to store pointer values. |
| true | Constant | Boolean literal true. |
| false | Constant | Boolean literal false. |
| iota | Constant | Special identifier for successive untyped integer constants in const blocks. |
| nil | Constant | Zero value for pointers, interfaces, maps, slices, channels, and function types. |
| append | Function | Built-in to append elements to a slice. |
| cap | Function | Returns capacity of slice, array, or channel. |
| close | Function | Closes a channel, signaling no more sends. |
| complex | Function | Constructs a complex number from real and imag parts. |
| copy | Function | Copies elements from source slice into destination slice. |
| delete | Function | Removes a key from a map or frees an entire dynamic memory object. |
| imag | Function | Returns the imaginary part of a complex number. |
| len | Function | Returns length of string, slice, array, map, or channel buffer. |
| make | Function | Allocates and initializes slices, maps, or channels. |
| new | Function | Allocates memory for a value of the given type. |
| panic | Function | Stops execution of the current goroutine and begins panicking. |
| print | Function | Low-level debugging print (not for production). |
| println | Function | Like print but adds spaces and newline. |
| real | Function | Returns the real part of a complex number. |
| recover | Function | Regains control of a panicking goroutine within a deferred function. |
| set | Type | A simple set<T> type. |
| Event<T> | Interface | Represents an Event object wrapping an object of type T |
| Error<E> | Interface | Represents an Error object wrapping an Error<Event<T>> object. |
| latest | Sentinel | Represents the most recent version of a package in the local environment. |

### Literals

#### Generally

AMI supports the same literal forms as Go—integers (decimal, octal, hexadecimal, binary), floating-point numbers with optional exponents, imaginary numbers, runes, interpreted and raw strings, Booleans, and composite literals for arrays, slices, maps, and structs. Literals have the same lexical rules and escape semantics as in Go; POP imposes no additional lexical constraints on literal formation.

#### Numeric Literal Formatting

Numeric literals admit underscores (\_) as digit separators to improve readability. Separators do not affect value and may not appear at the start or end of a number, adjacent to a base prefix (0x, 0o, 0b), adjacent to a decimal point, or in the exponent marker; otherwise, they are permitted freely within the digits of decimal, binary, octal, and hexadecimal forms.

Table 4: Literal Formatting

|  |  |  |
| --- | --- | --- |
| Category | Example(s) | Description |
| Integer | 42, 0600, 0xBadFace, 0b101 | Whole numbers. Can be decimal, octal (leading 0), hexadecimal (0x), or binary (0b). |
| Floating-point | 3.14159, 6.022e23,  1e-9 | Numbers with a decimal point or exponent. |
| Imaginary | 4i, 2.3e-4i | Complex number imaginary parts (real+imag parts use complex function). |
| Rune | 'a', '\n',  '\u03A9', '\x41' | A single Unicode code point, enclosed in single quotes. |
| String | "hello", "line1\nline2" | Interpreted string literals, support escape sequences. |
| Raw string | `hello\nworld` | Enclosed in backticks, preserves contents literally (no escapes). |
| Boolean | true, false | Predeclared identifiers of type bool. |
| Composite | [3]int{1, 2, 3}, map[string]int{"a":1},  struct{X int}{10} | Literal values of arrays, slices, maps, and structs. |
| Nil | nil | Predeclared identifier for zero value of pointers, interfaces, slices, maps, channels, and function types. |
| Type | []byte, string, MyPkg.Type | Defines data types. |

### Operators, Symbols, and Delimiters

Operators and delimiters mirror Go’s surface syntax (arithmetic, bitwise, logical, comparison, assignment, indexing, selection, grouping, and channel arrows). AMI retains ‘\*’ and ‘&’ symbols but interprets them over abstract memory objects rather than raw memory addresses; programs never observe concrete machine addresses (see §**Error! Reference source not found.**). Consequently, address-taking and dereference remain type-checked operations on opaque references, preserving RAII ownership while preventing aliasing to unmanaged memory. No new operator tokens are introduced for POP; pipeline constructs are expressed through declarative forms rather than operator overloading.

Table 5: Operators, Symbols, and Delimiters

|  |  |  |
| --- | --- | --- |
| Symbol | Category | Purpose / Meaning |
| + | Arithmetic | Addition (numbers) or concatenation (strings). |
| - | Arithmetic | Subtraction or unary negation. |
| \* | Arithmetic | Multiplication, or pointer dereference. |
| / | Arithmetic | Division. |
| % | Arithmetic | Modulus (remainder). |
| & | Bitwise / Address | Bitwise AND operator. |
| ^ | Bitwise | Bitwise XOR, or bitwise NOT (when unary). |
| << | Bitwise | Left shift. |
| >> | Bitwise | Right shift. |
| &^ | Bitwise | Bit clear (AND NOT). |
| += | Assignment | Add and assign. |
| -= | Assignment | Subtract and assign. |
| \*= | Assignment | Multiply and assign. |
| /= | Assignment | Divide and assign. |
| %= | Assignment | Modulus and assign. |
| &= | Assignment | Bitwise AND and assign. |
| = | Assignment | Assignment |
| ^= | Assignment | Bitwise XOR and assign. |
| <<= | Assignment | Left shift and assign. |
| >>= | Assignment | Right shift and assign. |
| &^= | Assignment | Bit clear and assign. |
| && | Logical | Logical AND (short-circuit). |
| ! | Logical | Logical NOT. |
| == | Comparison | Equality check. |
| != | Comparison | Inequality check. |
| < | Comparison | Less than. |
| <= | Comparison | Less than or equal. |
| > | Comparison | Greater than. |
| >= | Comparison | Greater than or equal. |
| = | Assignment | Simple assignment. |
| := | Short declaration | Declare and initialize variable in one step. |
| ... | Special | Variadic parameter in functions, or slice expansion. |
| \* (prefix) | Pointer | Dereference pointer. |
| & (prefix) | Pointer | Address-of operator. |
| <- | Channel | Send (chan <- val) or receive (val <- chan) operator. |
| ++ | Inc/Dec | Increment (statement, not expression). |
| -- | Inc/Dec | Decrement (statement, not expression). |
| ( ) | Grouping | Group expressions or function call arguments. |
| [ ] | Index/Slice | Index array, slice, map, or slice expressions. |
| { } | Block | Denote code blocks, struct/array literals, map literals. |
| <> | Generic type delimiter | Enclose type parameters and type arguments. Examples: func Map<T,U>(x T) U, Event<Int>, Result<T, E>. No whitespace is required but allowed: map<T, U>. Nested generics permitted: map<String, List<Int>>. |
| , | Separator | Separate elements in list, multiple return values, etc. |
| ; | Separator | Optional statement terminator (inserted automatically by lexer). |
| . | Selector | Field or method selector. |
| : | Special | Used in labels and composite literals (key: value). |
| -> |  | Not valid in Go (used in some other languages). |

### Comments and Whitespace

Whitespace (spaces, tabs, newlines) separates tokens and otherwise has no semantic effect. Line comments use // and block comments use /\* … \*/. As in Go, newlines may be significant to automatic semicolon insertion; the lexer applies the same insertion rules, ensuring familiar formatting behavior while leaving POP’s declarative grammar unaffected.

Code 4: AMI Comment Examples

|  |
| --- |
| //  /\*  . . . Block Comment . . .  \*/ |

### The Special Underscore (**\_**) Identifier

The underscore (\_) is a reserved, write-only identifier used to intentionally discard values and to mark unused positions in declarations and calls. It is not a bindable name and has no address; it cannot be referenced on the right-hand side of an expression, captured, exported, or shadowed. Its purpose is to make programmer intent explicit, eliminate “unused” diagnostics, and enable aggressive compile-time and code-generation optimizations consistent with POP’s analyzability and immutability guarantees.

**Discarding results.** In assignments and multi-value returns, the underscore (\_) on the left-hand side signals that the corresponding value is intentionally ignored. The compiler treats this as a strong hint to avoid materializing copies for the discarded value, which is particularly important in AMI’s copy-semantics model.

**Unused parameters (declarations).** Within function parameter lists, naming a parameter using the underscore ( \_) declares that the argument is required for arity and type but will not be used by the callee, preventing unused-parameter errors and permitting dead-code elimination around that symbol. This is particularly useful when implementing abstractions in code.

**Prohibitions and constraints.** The underscore (\_) cannot be assigned to a variable, addressed, stored, or propagated across scope boundaries; it cannot be used as a field name, method receiver, package name, or identifier in exported declarations. Its role is limited to syntactic positions that discard or suppress use. These constraints ensure that the underscore (\_) remains a pure compile-time signal with no observable runtime behavior, aligning with AMI’s event immutability and effect-bounding rules.

### Package Identifiers

#### Package Names

Except as otherwise stated in this subsection, package names must meet the standards for all identifiers (§2.1.2). They may be uppercase, lowercase or mixed case-sensitive strings. Package names must use only ASCII characters.

#### Version Tags

AMI requires explicit, analyzable package versions to ensure reproducible builds and safe composition. Every non-main package declares its version in the package clause; imports constrain acceptable versions at the call site. This mirrors the source-level versioning model introduced in the example code (§2.2):

Code 5: AMI source package statement example

|  |
| --- |
| package mything:0.0.1 |

and the import constraint syntax with comparison and Boolean operators. For example:

Code 6: package versioning rules

|  |
| --- |
| >=v0.0.1  > v0.0.1 && != v0.0.2 |

**Normative form.** AMI adopts Semantic Versioning 2.0.0 for identifiers (MAJOR.MINOR.PATCH[-PRERELEASE|githash][+BUILD]) (Preston-Werner, n.d.). In package declarations the leading v is optional; in import constraints, an optional leading v is accepted for ergonomics (e.g., >=v1.4.2). Projects may also use the sentinel identifier latest in imports; it is resolved to a concrete version by the workspace at compile time.

**Validation regex (PCRE/RE2-compatible).** Use this fully anchored pattern for tokens that appear in import constraints (allows optional v, standard prerelease, and build metadata such as short/long git hashes):

Code 7: Package SemVer Regular Expression

|  |
| --- |
| ^(?:v)?(0|[1-9]\d\*)\.(0|[1-9]\d\*)\.(0|[1-9]\d\*)  (?:-((?:0|[1-9]\d\*|[A-Za-z-][0-9A-Za-z-]\*)  (?:\.(?:0|[1-9]\d\*|[A-Za-z-][0-9A-Za-z-]\*))\*))?  (?:\+([0-9A-Za-z-]+(?:\.[0-9A-Za-z-]+)\*))?$ |

For the package clause, reject a leading v by matching the same pattern without the (?:v)? prefix.

**Resolution semantics.** Version constraints are bound at **compile time**. The compiler resolves latest against the workspace and rejects incompatible sets (e.g., conflicting constraints across imports) before code generation, preserving POP’s analyzability and preventing runtime drift.

Code 8: Package definition and imports example

|  |
| --- |
| package mything:0.0.1 // valid (no leading v)  import github.com/acme/csv >=v0.0.1 // ergonomic "v" accepted in imports  import github.com/acme/db == latest // workspace-resolved alias  import github.com/acme/x > v0.0.1 && != v0.0.2 |

Further discussion of packaging and imports can be found in §**Error! Reference source not found.**.

### Compiler Directives (Pragmas)

A compiler directive is a line-oriented annotation that influences compilation without changing program semantics at runtime. The canonical form is:

Code 9: Pragma Pattern

|  |
| --- |
| //@ami:<directive> [arg[, arg…]] |

Directives apply to the immediately following declaration or, when placed at file top before any declarations, to the compilation unit. Unknown directives MUST be ignored with a warning. Directives MUST NOT enable capabilities, I/O, or concurrency beyond what is possible in ordinary code; they are limited to build selection, diagnostics, layout hints, and other analyzable actions so as not to violate POP guarantees. Toolchains may define additional //@tool:<name> directives; such directives are implementation-defined and non-portable.

Compiler directives do not override directives found in the project ami.workspace file (§2.2.1).

## Constructing Pipelines by Example

This section walks through a minimal AMI project we’ll reuse in later sections: a workspace manifest (ami.workspace) and a source file (main.ami).

### Project Manifest (**ami.workspace**)

Code 10: Example ami.workspace

|  |
| --- |
| ---  # ami.workspace (UTF-8 YAML)  version: 1.0.0 # schema version (SemVer)  toolchain:  compiler:  concurrency: 16 #NUM\_CPU is a reserved macro  target: ./build  env:  - os: windows/amd64  - os: linux/amd64  - os: linux/arm64  - os: darwin/amd64  linker: {}  linter: {}  packages:  - main:  version: 0.0.1  license: MIT  root: ./src  import:  - github.com/asymmetric-effort/ami/stdlib/file >= v0.0.1  - github.com/asymmetric-effort/ami/stdlib/csv > v0.0.1  - github.com/asymmetric-effort/ami/pgsql/db ==latest  - ./someProject > v0.0.1 # local subproject  - someProjectVariantA:  alias: someProject  version: 0.1.3-prerelease+g1234abcd  root: ./subtrees/someProject/v0\_1\_3 |

#### **Purpose and scope**

Every AMI project must include a UTF-8 YAML file named ami.workspace at the repository root. The manifest declares the build matrix, toolchain options, packages (including main), and dependency rules. File-level directives MAY narrow—but MUST NOT widen—dependencies resolved from ami.workspace. On conflict, the manifest prevails.

Every AMI project MUST contain a UTF-8 encoded YAML file named **ami.workspace** at the repository root. This manifest declares the build matrix, toolchain options, packages that comprise the workspace (including the main package), and all dependency rules. File-level compiler directives MAY narrow but MUST NOT widen dependency rules resolved from ami.workspace. If a conflict exists, the manifest prevails.

#### **File name and location**

The file name is **exactly** ami.workspace. It resides at the workspace root of the repository.

#### Top-level schema

The manifest has these top-level sections:

* version (schema version; SemVer)
* toolchain (compiler/linker/linter configuration and build matrix)
* packages (ordered list of package entries, starting with main)

#### Toolchain

toolchain.compiler MAY declare—

1. concurrency (compile parallelism),
2. output target directory,
3. and a cross-compile build matrix via env entries of the form {os: "<os>/<arch>"}.
4. Linker/linter settings MAY be declared analogously.

The example values in the draft (Windows/Linux/macOS, amd64/arm64) remain valid.

#### Concurrency Macro (NUM\_CPU)

The toolchain.compiler.concurrency parameter may be set to any positive integer, or the programmer may use the NUM\_CPU:

Code 11: ami.workspace toolchain.\*

|  |
| --- |
| toolchain:  compiler:  concurrency: NUM\_CPU  target: ./build |

When NUM\_CPU is used, the compiler autodetects the host CPU count and sets concurrency accordingly.

#### Packages

packages is an ordered list. The first entry MUST be main, which defines the project’s executable entry package:

* main.version is the project version (SemVer).
* main.root is the path (relative to the workspace root) where main.ami resides.
* main.import lists dependency declarations (see §2.2.1.6).

Additional entries are subprojects that the workspace builds alongside main. Subprojects MAY specify alias to allow multiple versions of the same logical package to coexist, each with a distinct source tree. This enables fast experiments and migrations between local project versions.

### Source File (**main.ami**)

Below is an example main.ami source file demonstrating an end-to-end Pipeline-Oriented Programming (POP) example in the AMI language:

Code 12: Example AMI Source

|  |
| --- |
| // file: main.ami // (c) 2025 Asymmetric-Effort, LLC.  package main:0.0.1 // // We must import dependencies after package and before anything else // import github.com/asymmetric-effort/ami/stdlib/file >= v0.0.1 import github.com/asymmetric-effort/ami/stdlib/csv > v0.0.1 import github.com/asymmetric-effort/ami/postgresql/db == latest import github.com/asymmetric-effort/someProject > v0.0.1 && != v0.0.2  // // The following lines define a declarative pipeline // Ingress(  //  // Each Ingress node heads a pipeline and may be named.  // Names are optional but recommended for all node types.  //  name=csvReaderPipeline,  //  // The ingress node is triggered by the FsNotify resource  // of the file package on file.WriteEvent, which will monitor  // events on the defined source (/opt/input.csv).  // (Ingress.in is a SourceSpec)  //  in=file.FsNotify(  event=file.WriteEvent,  source="/opt/input.csv",  ),  //  // On an event, the worker function will be executed, passing Event<T>  // as input.  //  worker=csv.Load,  //  // The node will execute the worker function with minWorker instances  // at minimum, and maxWorkers as load requires.  //  minWorkers=2,  maxWorkers=8,  //  // On any unhandled error the worker function returns to the node level,  // that error will be forwarded to an error handling pipeline. By default,  // this would be ErrorPipeline(). But it could be another.  //  onError=ErrorPipeline,  //  // At compile time we ensure the Ingress node only grants specific  // capabilities at compile time with runtime limits for granular permissions.  //  capabilities=[FileRead:"/opt/input.csv"],  //  // The event returned by the worker function will be pushed to the downstream  // edge.  //  // This type attribute helps the compiler guarantee no conflict by defining  // Event<T> as Event<[]byte>  type=[]byte, ).Transform(  //  // The transformer node consumes events at its upstream edge as either a FIFO  // or LIFO channel with a minimum and maximum capacity. When capacity is  // reached, the backpressure policy is enforced. We notice that the edge is  // strongly typed (e.g., []byte). This is the type T of each Event<T>.  // Edge binding (consumer .in is an EdgeSpec)  //  in=edge.FIFO(  minCapacity=10,  maxCapacity=20,  backpressure=block,  type=[]byte,  ),  //  // Each event received across the edge boundary is passed as Event<T> to  // the worker function (e.g., CleanFunc)  //  worker=someProject.CleanFunc,  //  // The worker functions execute in a number of virtual machines, where the  // minWorkers run at all time up to maxWorkers when backpressure events  // will start.  //  minWorkers=2,  maxWorkers=8,  //  // The event returned by the worker function will be pushed to the downstream  // edge.  //  onError=ErrorPipeline,  // This type attribute helps the compiler guarantee no conflict by defining  // Event<T> as Event<someProject.CsvRecord>  type=someProject.CsvRecord, ).FanOut(  //  // A Fanout node has a primary path (direct downstream) and other concurrent  // paths. Events are pulled from the upstream into an edge.FIFO() or edge.LIFO()  // edge. In this example, the type is CsvRecord (a struct)  //  in=edge.FIFO(  minCapacity=10,  maxCapacity=20,  backpressure=block,  type=someProject.CsvRecord,  ),  //  // A FanOut node has no user-defined worker function. The worker function is  // created by the compiler to receive and broadcast Events. But the node can  // control traffic flow through the minWorkers and maxWorkers attributes.  //  minWorkers=2,  maxWorkers=8,  //  // The default downstream pipeline and any pipelines defined in the `out`  // list will receive a copy of Event<U> returned by the worker function.  //  out=[  Logger.LogPipeline,  ],  //  // If an error occurs processing events along the pipeline, the error  // will be pushed onto the pipeline defined by OnError.  //  onError=ErrorPipeline,  type=someProject.CsvRecord, ).Collect(  //  // A Collect node is the inverse of FanOut. It collects Event<T> data  // from multiple input pathways, orders them and forwards them on to the  // downstream pipeline. The 'in' attribute of .Collect() uses edge.MultiPath()  // to aggregate inputs from multiple pipelines into a single flow. The  // input types must be compatible.  //  in=edge.MultiPath(  inputs=[  //  // Each path is a separate pipeline input, either from the upstream  // pipeline segment or from a different pipeline altogether.  //  edge.FIFO(  minCapacity=10,  maxCapacity=20,  backpressure=block,  type=someProject.CsvRecord,  ),  //  // An 'edge.Pipeline()' represents the FanOut() of another  // pipeline. On this receiving end of this edge, we act as  // we have with all other inbound edges.  //  edge.Pipeline(  name=otherUpstreamPipeline,  minCapacity=10,  maxCapacity=20,  backpressure=dropNewest,  type=someProject.CsvRecord,  ),  ],  //  // As Event<T> objects flow into a Collect node, they are queued,  // sorted, then forwarded to the downstream pipeline segment.  //  merge=Sort(  algorithm=ascendingTimeStamp,  window=120,  key=event.Created,  ),  ),  //  // Flow into the collect node is controlled by minWorkers and maxWorkers.  // The workers are responsible for performing the merger of event streams.  //  minWorkers=2,  maxWorkers=8,  //  // Input flows may have compatible abstract types. But the collect node  // will ensure that the output is a specific concrete type. This is used  // by the compiler to guarantee no conflicts exist.  //  type=someProject.CsvRecord, ).Egress(  //  // An Egress node delivers output to some external sink. It accepts Event<T>  // on an edge like other nodes, then processes it to the target sink using  // its fleet of workers.  //  in=edge.FIFO(  minCapacity=10,  maxCapacity=20,  backpressure=block,  type=someProject.CsvRecord,  ),  //  // The worker function may use the node-state table to share configuration  // when connecting to an external system.  //  worker=db.ExecStoredProc(  proc="csvRecordInsert",  conn=dbConnection(  //  // config is a configuration pipeline  // which reads a config file and terminates  // at an Egress() node, where the .get()  // method of Egress() allows pipelines to  // pop values from the configuration as a  // key-value store.  dbHost=config.Pipeline("db\_config").get("dbHost"),  dbPort=config.Pipeline("db\_config").get("dbPort"),  dbName=config.Pipeline("db\_config").get("dbName"),  dbUser=config.Pipeline("db\_config").get("dbUser"),  dbPass=config.Pipeline("db\_config").get("dbPass"),  ),  ), // factory returns a worker  minWorkers=2,  maxWorkers=8,  //  // Like ingress nodes, capabilities provide granular access control.  //  capabilities=[  NetOut: "db.acme-corp:5432",  ],  ) |

We will use this code example to analyze how the AMI programming language works. This will help the reader to digest the language and observe the benefits of Pipeline-Oriented Programming (POP).

### Package Declaration

Every AMI source file begins with a package clause. Each project defines a main package as the build entry point. In library projects, the main package file may be empty but MUST exist. Subprojects imported by a project with a main package do not define their own main; they are compiled only through the consuming project.

#### Package Identifier Rules

Package names obey the rules defined in §2.1.9.

#### Versioning

All packages MUST declare an explicit semantic version in the package clause (§2.1.9.2):

Code 13: AMI Package declaration pattern

|  |
| --- |
| package <name>:<MAJOR.MINOR.PATCH[-PRERELEASE][+BUILD]> |

For example,

Code 14: Example AMI Package Declaration

|  |
| --- |
| package mything:0.0.1 |

Versioning in source is mandatory for reproducible builds and consistent composition. Files without a version do not compile, and such packages cannot be imported.

AMI is a source-distribution language: packages ship as source, not prebuilt artifacts. Embedding the version in the declaration prevents configuration drift and aligns with POP’s emphasis on static analyzability. The version token follows Semantic Versioning 2.0.0. In package clauses, a leading v is optional (e.g., package foo:1.2.3 or package foo:v1.2.3). Import constraints may also accept an optional leading v for ergonomics; the compiler normalizes these and resolves them against the workspace manifest (see §2.1.9.2).

#### Importing Packages

An AMI source file declares imports immediately after the package clause and before any other declarations. Each import names a package and supplies a version constraint; the compiler resolves constraints to a concrete SemVer at build time, ensuring deterministic, analyzable dependencies.

**Form and placement.** Imports are line-oriented and appear as:

Code 15: AMI Import Pattern

|  |
| --- |
| import <module-path> <constraint> |

Examples (spacing normalized):

Code 16: AMI Import Examples

|  |
| --- |
| import github.com/asymmetric-effort/ami/stdlib/file >= v0.0.1  import github.com/asymmetric-effort/ami/stdlib/csv > v0.0.1  import github.com/asymmetric-effort/ami/postgresql/db == latest  import github.com/asymmetric-effort/someProject > v0.0.1 && != v0.0.2 |

Imports are line-oriented and must immediately follow the package line.

**Version constraints and determinism.** Constraints use >=, >, ==, != and may be combined with &&, ||, !. The ergonomic v prefix is permitted in import constraints (e.g., v0.0.1). The symbolic selector latest is allowed but is resolved to a specific version before code generation; unresolved or conflicting constraints are rejected.

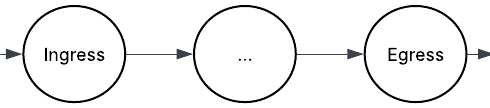
**Workspace interaction.** The workspace (ami.workspace) governs version sources and mappings (aliases, local subprojects). Relative imports must refer to subprojects declared in the same workspace; parent-directory (..) traversal is prohibited. Thus, ./someProject is legal only if that subproject is declared in ami.workspace.

**SemVer normalization.** Package-clause policy: a leading v is **optional** (see §2.2.3). Import constraints may also include it. The compiler normalizes and validates all versions against the SemVer grammar prior to resolution.

**Rationale.** Binding imports to concrete versions at compile time preserves POP’s analyzability and repeatability, yielding a static, auditable dependency graph and preventing non-determinism from floating versions.

### Node-Chained Pipeline Notation

Figure 14: A Simple Pipeline Graph Visualization



AMI expresses pipelines with a node-chained notation for linear segments from ingress to egress:

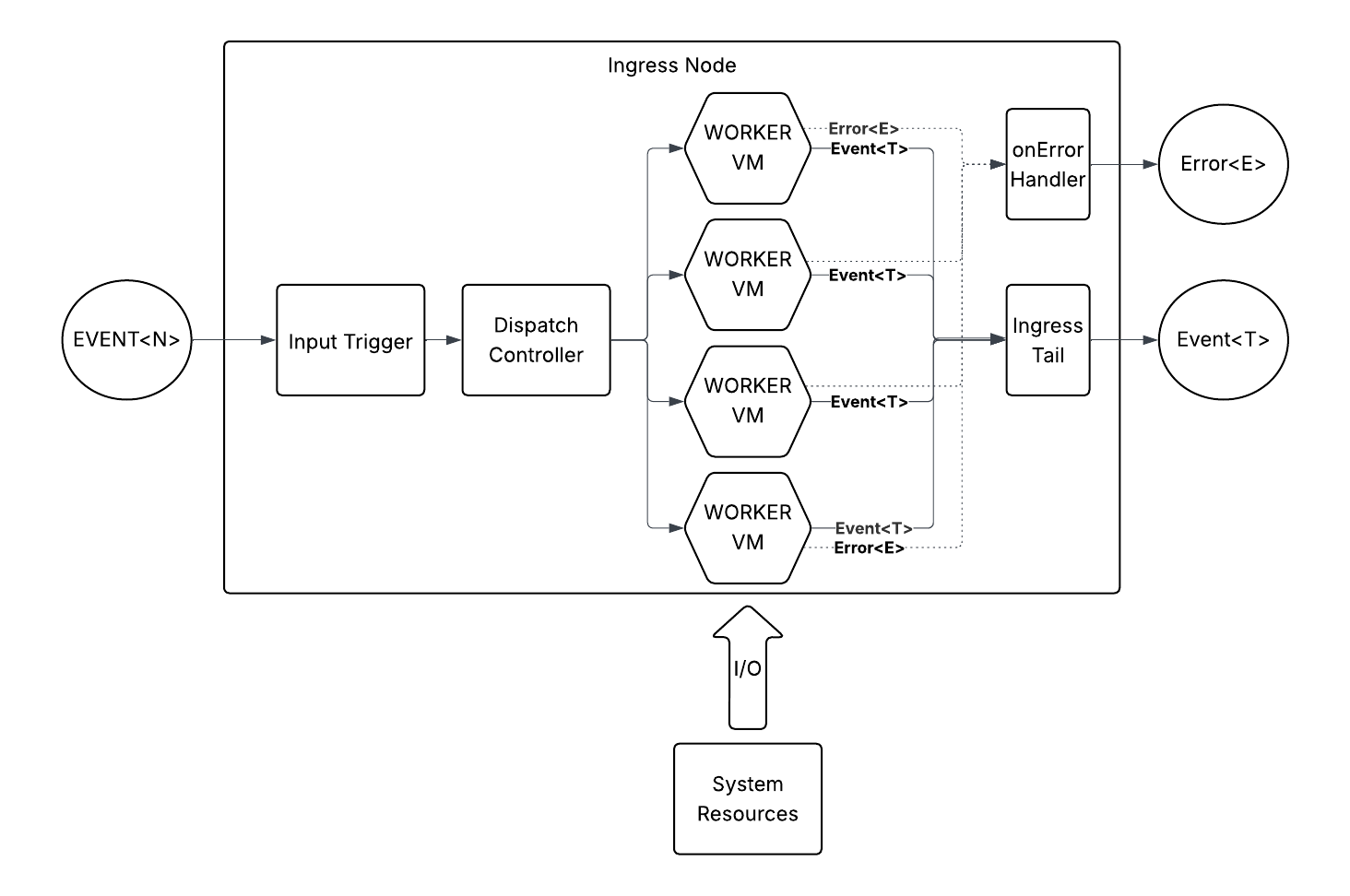
Figure 15: Node-chained pipeline example

|  |
| --- |
| Ingress().Transform().Egress() |

Here, Ingress(), Transform(), and Egress() are pipeline nodes, and the dot (.) denotes the edge connecting adjacent nodes. Because POP/AMI adopt graph-theoretic terminology, we describe pipelines as graphs—nodes connected by edges.

### Ingress Node

Figure 16: Ingress Node Topology



An Ingress() node heads a pipeline and, via its name, identifies it. It is the only node allowed to acquire data from outside the program’s declarative graph, establishing the functional entry point, pipeline identity, and the security boundary for external I/O. In the running example, the node subscribes to a filesystem notification source and spawns a worker that converts each notification into a strongly typed event.

Below is an example Ingress() node:

Code 17: AMI Ingress Node Example

|  |
| --- |
| Ingress(  name=csvReaderPipeline,  in=file.FsNotify(  event=file.WriteEvent,  source="/opt/input.csv",  ),  worker=csv.Load,  minWorkers=2,  maxWorkers=8,  onError=ErrorPipeline,  capabilities=[FileRead:"/opt/input.csv"],  type=[]byte, ) |

#### **Required Attributes.**

An Ingress() declaration may specify the following attributes:

Table 6: AMI Ingress Node Required Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Attribute | Type | Required | Description / Notes |
| in | SourceSpec | Yes | Binds the node to an external trigger (e.g., file.FsNotify( event=file.WriteEvent, source="/opt/input.csv")). The compiler consumes this spec to generate the glue code that receives the trigger and constructs the input event, ensuring the binding is analyzable at compile time. |
| worker | WorkerFuncSpec | Yes | Identifies the worker function which will run in a sandboxed virtual machine to perform the Ingress I/O operations needed to consume raw data and emit an Event<T> and/or Error<E> object. |
| minWorkers | uint | Yes | lower and upper bounds on the pool of warm worker VMs the runtime maintains. The controller dispatches a new VM when needed up to maxWorkers. Ordering is not guaranteed by the Ingress itself; downstream nodes should impose ordering if required. |
| maxWorkers | uint | Yes |
| onError | PipelineName | No | Binds node to a pipeline to which all Error<E> events will be pushed. |
| capabilities | List<cap> | Yes | A compile-time declared set of least-privilege permissions that gate I/O (e.g., FileRead:"/opt/input.csv"). The compiler uses these to specialize the worker VM and the runtime enforces them at the OS boundary. |
| type | Data type | Yes | The concrete payload type T of Event<T> produced by the worker. This attribute pins the downstream type contract and allows the compiler to resolve any residual ambiguities in the node’s type flow. |
| name | string | No | A unique identifier representing the ingress and the entire pipeline. When the entire pipeline is referenced logically, it uses this name (even when addressing pipeline output from an Egress() node.) |

#### **Pipeline Identity and Analyzability.**

If name is omitted, the compiler assigns a UUID to name (Leach, Mealling, & Salz, 2005; International Organization for Standardization, 2014). name contributes to EventId construction, providing stable pipeline-wide traceability. Binding to a concrete source via in and a concrete payload via type preserves POP’s determinism and compile-time analyzability.

#### **Execution Model.**

On a trigger from the SourceSpec, the runtime controller selects or provisions a worker VM within [minWorkers, maxWorkers] and invokes it with Event<N>. The worker returns either a non-nil Event<T>, which the node forwards downstream, or nil to indicate no emission. Any Error<E> is routed to onError without halting other event processing.

#### **Security Boundary.**

Because Ingress() is where external data first enters the POP graph, it MUST declare all I/O via capabilities. These declarations are verified at compile time and enforced at runtime, ensuring acquisition is auditable and limited to the minimum necessary scope.

Code 18: Ingress/Egress capabilities clause example

|  |
| --- |
| capabilities=[  FileRead:"/opt/input.csv"  ], |

capabilities is a list of key-value pairs. The key names the general capability (e.g., FileRead); the value scopes it to specific resources or patterns (e.g., the file path). These declarations enable the compiler to dead-strip unauthorized I/O paths and the runtime to enforce guards so I/O occurs only on explicitly allowed resources.

#### **Sandboxed Worker Virtual Machines.**

Workers execute inside capability-bounded virtual machines. Side effects (I/O, OS interaction) are permitted only where the node’s capabilities authorize them (e.g., Ingress/Egress may perform I/O; Transform remains pure). Capabilities §2.2.5.4) are enforced at **compile time** (VM shaping) and **runtime** (policy checks).

Each sandboxed worker VM is a **specialization unit**: the compiler builds a minimal VM for the specific worker function and optimizes it accordingly. These VMs are **not** general-purpose sandboxes; they exist solely for their bound worker, though common functionality may be reused across VMs.

### Worker Function (WorkerFuncSpec) Signature and Semantics

A **worker** is any function bound to an Ingress, Transform, or Egress node that implements POP’s uniform contract:

Code 19: AMI Worker Function Signature

|  |
| --- |
| func <identifier>(in Event<N>) (out Event<T>, err Error<E>) |

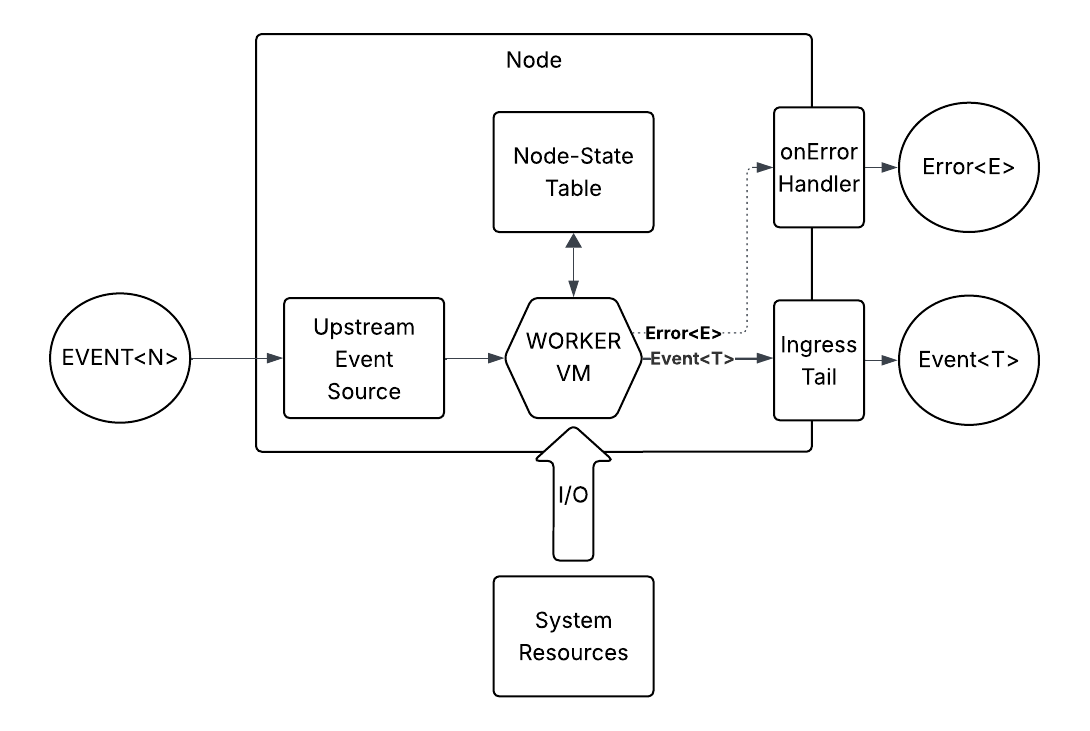
This signature is mandatory. Event<N> is the node’s input type; the returned Event<T> MUST match the node’s declared downstream type; Error<E> is the typed diagnostic routed to the node’s error pipeline.

#### **Compile-time Guarantees.**

The compiler verifies that T matches the node’s declared downstream type and that edge type-compatibility and capacity contracts are satisfiable across node boundaries. These checks produce an analyzable plan for memory, ordering, and backpressure before code generation. Any ambiguity in edge types is a compile-time error.

#### **Runtime Dispatch.**

Figure 17: Ingress Runtime Dispatch



On arrival of Event<N> at the ingress source, the compiler-generated controller dispatches to an idle worker VM or elastically provisions one up to maxWorkers. Dispatch is non-blocking: once the worker is launched, the controller immediately handles the next message.

Each worker is wrapped by compiler-generated supervision (restart/retry, error handling, output routing). On return, a non-nil out (Event<T>) is delivered to the downstream edge; any err (Error<E>) is routed to onError.

#### **Single-Disposition Rule.**

Each invocation must result in exactly one of the following dispositions:

Table 7: Worker Function Dispositions

|  |  |  |  |
| --- | --- | --- | --- |
| Disposition | out | err | Comments |
| Success with emission | out != nil | err == nil | one Event<T> is forwarded. |
| Success without emission (“silence”) | out == nil | err == nil | no message is enqueued; this is a deliberate drop (e.g., filtering). |
| Failure | out == nil | err != nil | no downstream event; the error is routed to the node’s error pipeline. |
| Forbidden | out != nil | err != nil | rejected by the compiler |

#### Worker Function **Factories**

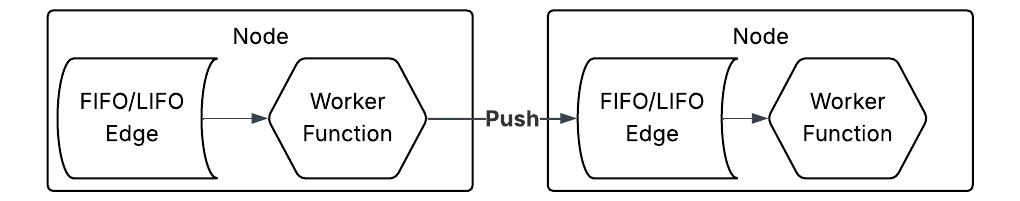
A node’s worker may be supplied by a **factory** that returns a worker with captured configuration (e.g., db.ExecStoredProc(...)). Constant arguments are resolved at compile time when possible. The compiler may specialize and inline the result, effectively eliminating the factory.

#### **Immutability and Locality.**

Memory is not shared across worker VM boundaries. Inputs are immutable; the runtime copies them into the worker. Outputs are immutable copies tagged as Event<T> or Error<E> and enqueued to their respective queues.

### Edges

Figure 18: Edge Anatomy



An **edge** connects two nodes and is always defined on the **consumer** side. The consumer specifies capacity, a backpressure policy, and the storage algorithm (FIFO or LIFO). This edge definition is the EdgeSpecifier—

Code 20: Ingress node 'in' attribute example

|  |
| --- |
| in=edge.FIFO(  minCapacity=10,  maxCapacity=20,  backpressure=block,  type=someProject.CsvRecord,  ), |

We use this pattern in Transform(), FanOut(), Collect(), and Egress() nodes (discussed in §2.2.8, §2.2.9 , §2.2.10, §2.2.11).

#### EdgeSpecifier Attributes

Any EdgeSpecifier must define at least these four attributes:

Table 8: EdgeSpecifier Attributes

|  |  |  |
| --- | --- | --- |
| Attribute | Type | Description |
| minCapacity | uint | Minimum number of memory spaces for an edge at all times. |
| maxCapacity | uint | Maximum number of memory spaces for an edge under load. |
| backpressure | uint | A policy which defines the edge behavior when load exceeds the maximum capacity of the edge. |
| type | Data Type | Declares the data type supported by the edge structure. |

#### Edge Algorithms (FIFO, LIFO) and Capacity

An edge operates in first-in, first-out (FIFO) or last-in, first-out (LIFO) mode. Its size (**capacity**) is the number of elements it can store. On initialization, the runtime allocates minCapacity slots; under load it grows up to maxCapacity before enforcing the configured backpressure policy. These constraints are checked at compile time and enforced at runtime.

Actual memory for an edge is:

minCapacity and maxCapacity are **counts**, not bytes. The compiler uses the type attribute to determine T in Event<T> and thus to compute sizes.

#### Edge Methods

An edge is a compiler-generated, opaque object defined by its attributes. It exposes a small API to both the upstream (producer) and downstream (consumer) workers:

Table 9: Worker Interface Edge Methods

|  |  |  |
| --- | --- | --- |
| Method | Return | Description |
| .capacity | uint | Returns the actual capacity of the edge. |
| .minCapacity() | uint | Returns the minimum capacity of the edge. |
| .maxCapacity() | uint | Returns the maximum capacity of the edge. |
| .count() | uint | Returns the number of items in the edge. |

These methods are available to the worker function.

#### Event Flow from Upstream Worker into the Edge

A worker returns two values: Event<T> and Error<E>. The compiler-generated wrapper routes any non-nil Event<T> to the downstream edge and any non-nil Error<E> to onError. A nil Event<T> simply means **no emission** (nothing is enqueued). This aligns with the single-disposition rule (§2.2.6.3).

### Transform Node

Figure 19: Transform Node Topology

A diagram of a server

AI-generated content may be incorrect.

Transform() executes user-defined computation to transform events. Unlike Ingress()/Egress(), it is **pure** (no I/O or syscalls); side effects are disallowed by capabilities policy.

Code 21: AMI Transform Node Example

|  |
| --- |
| ...).Transform(  in=edge.FIFO(. . .),  worker=myproject.CleanFunc,  minWorkers=2,  maxWorkers=8,  onError=ErrorPipeline,  type=someProject.CsvRecord, )... |

* **Edge (consumer side):**

in takes an **EdgeSpecifier**; see **§2.2.7 Edges** for the form and semantics, **§2.2.7.2** for minCapacity, maxCapacity, backpressure, type, and **§2.2.7.3** for edge methods.

* **Worker:**

Must implement the uniform signature; the compile-time checks and single-disposition rule from **§2.2.6** apply unchanged.

* **Execution & scaling:**

minWorkers/maxWorkers control the pool; onError routes Error<E>; behavior mirrors the Ingress execution model (dispatch is non-blocking).

* **Type contract:**

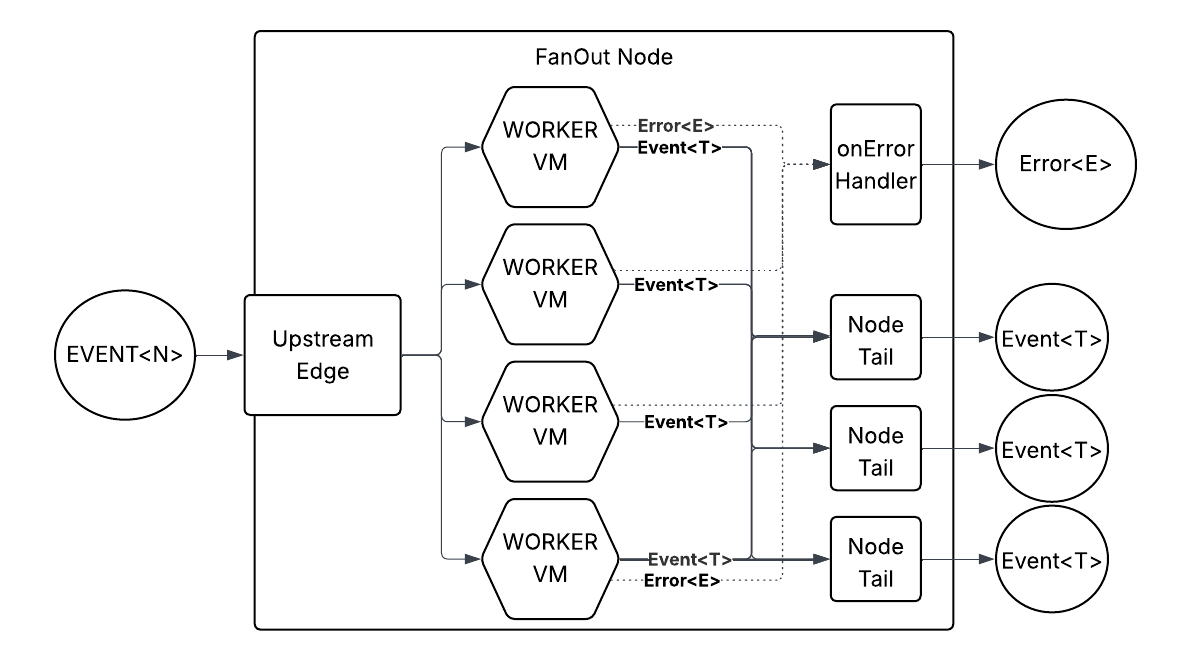
type declares the produced Event<T> payload; same contract used in **Ingress** (§2.2.5, attribute type).

* **No I/O:**

By capability policy, Transform() has no external capabilities; contrast with **§2.2.5.4 Security Boundary** for Ingress().

### FanOut Node

Figure 20: FanOut Node Topology



The FanOut() node, like the Collect() node is purely declarative construct. The programmer cannot provide user-defined worker functions to operate in these two specialized event router node types. Nonetheless, these two node types are powerful tools when developing pipeline-oriented software using POP. The FanOut() node allows an Event<T> to be broadcast to many downstream pipeline segments. The Collect() node consolidates the Event<T> streams from many inputs into a single downstream.

Below is an example FanOut() node declaration:

Code 22: AMI FanOut node example

|  |
| --- |
| …).FanOut(  in=edge.FIFO(  minCapacity=10,  maxCapacity=20,  backpressure=block,  type=someProject.CsvRecord,  ),  minWorkers=2,  maxWorkers=8,  out=[  LoggerPipeline,  ],  onError=ErrorPipeline,  type=someProject.CsvRecord, ). … |

#### FanOut Semantics

**FanOut()** is a declarative broadcast router. It has **no user-defined worker**; the compiler synthesizes a worker whose sole job is to **dispatch each incoming** Event<T> **to all declared downstream pipelines**. Concurrency is controlled via minWorkers/maxWorkers (see §2.2.6). If any downstream edge uses backpressure=block, the FanOut() node blocks emission to **all** downstream edges; other policies allow continued delivery to unblocked paths.

The upstream interface is the usual EdgeSpecifier on in, e.g., edge.FIFO(...) or edge.LIFO(...) with minCapacity, maxCapacity, backpressure, and type (see §2.2.7). The node’s type fixes T for emitted events, and errors flow to onError.

Downstream edges are **fixed at compile time**. The default (“primary”) downstream is the next node in the **node-chained notation** (§2.2.4). Additional destinations are named in out=[ … ], where each identifier refers to the name of another pipeline’s Ingress() or Collect() node.

Minimal example (schematic):

Code 23: Minimal FanOut Schematic

|  |
| --- |
| ...).FanOut(  in=edge.FIFO(  minCapacity=10,  maxCapacity=20,  backpressure=block,  type=someProject.CsvRecord,  ),  minWorkers=2,  maxWorkers=8,  out=[  Logger.LogPipeline  ],  onError=ErrorPipeline,  type=someProject.CsvRecord,  )... |

Cross-refs covered: node-chained notation (§2.2.4), worker model & error routing (§2.2.6), edge specification & capacities (§2.2.7).

#### Backpressure Impact on FanOut Nodes

The FanOut() node’s worker is **compiler-generated** and runs in a specialized, capability-bounded VM. Its sole responsibility is to dispatch each upstream Event<T> to **all** configured downstream pipelines. If any downstream edge is configured with backpressure=block, a backpressure event **blocks** the FanOut() node from emitting to **any** downstream. With other backpressure policies, FanOut() continues delivering to unblocked downstream edges while the affected edge applies its policy.

#### Downstream Edges

The FanOut() node has a **primary** downstream defined by POP/AMI’s dot-chained syntax (treat this as edge[0]). Additional downstream edge is declared in the out attribute:

Code 24: FanOut 'out' attribute example

|  |
| --- |
| out=[  Logger.LogPipeline,  <PipelineIdentifier>, ], |

The out attribute may list zero or more pipelines. Each identifier (e.g., Logger.LogPipeline) must match the name of another pipeline’s Ingress() or Collect() node. Downstream edges are fixed **at compile time**, enabling compiler optimizations consistent with POP. A <PipelineIdentifier> uniquely identifies a pipeline, as defined by the name attribute of its Ingress() node.

### Collect Node

Figure 21: Collect Node Topology

A diagram of a server

AI-generated content may be incorrect.

Collect() consolidates multiple upstream Event<T> feeds into a single downstream edge—the inverse of FanOut() (§2.2.9). It is declarative (no user-defined worker). The consumer edge remains a single in attribute, but uses edge.MultiPath(...) to multiplex sources; each source is an EdgeSpecifier (§2.2.7).

Figure 22: Collect Data Flow

A diagram of a work flow

AI-generated content may be incorrect.

The Collect() node consolidates Event<T> feeds from multiple upstream edges into a single downstream edge. This is the inverse of the FanOut() node. Below is an example:

Code 25: AMI Collect node example

|  |
| --- |
| ).Collect(  in=edge.MultiPath(  inputs=[  edge.FIFO(  minCapacity=10,  maxCapacity=20,  backpressure=block,  type=someProject.CsvRecord,  ),  edge.Pipeline(  name=otherUpstreamPipeline,  minCapacity=10,  maxCapacity=20,  backpressure=dropNewest,  type=someProject.CsvRecord,  ),  ],  merge=Sort(  algorithm=ascendingTimeStamp,  window=120,  key=event.Created,  ),  ),  minWorkers=2,  maxWorkers=8,  onError=ErrorPipeline,  type=someProject.CsvRecord, ) |

The following table defines the attributes of the Collect() node:

Table 10: AMI Collect() node attributes

|  |  |  |
| --- | --- | --- |
| Attribute | Type | Notes |
| in | EdgeSpecifier | The in may include default upstream edges (e.g., edge.FIFO(...)) and named pipeline sources via edge.Pipeline(name=...). Each entry follows the same capacity/backpressure/type rules as other edges (§2.2.7.2). See also §2.2.10.1, §2.2.10.2. |
| merge | MergeSpecifier | merge=... defines how concurrent inputs are combined before delivery (e.g., Sort(algorithm=..., window=..., key=...)). Use this to impose ordering if needed; otherwise, delivery order is not guaranteed (§2.2.6 execution semantics). |
| minWorkers | uint | Minimum number of provisioned workers |
| maxWorkers | uint | Maximum number of provisioned workers |
| onError | <PipelineIdentifier> | Routes Error<E>. |
| type | <T> | type sets T for emitted events and MUST match source/edge compatibility checked at compile time (§2.2.6.1, §2.2.7). |

As with other nodes, the single-disposition rule applies (§2.2.6.3). Collect() nodes are also not allowed to perform external I/O operations; it routes and merges events only (contrast Ingress() capabilities in §2.2.5.4).

#### <EdgeSpecifier>: edge.MultiPath(…)

Figure 23: edge.MultiPath(...) topology

A diagram of a network

AI-generated content may be incorrect.

edge.MultiPath(...) is not mere syntactic sugar. It cleanly separates the consumeredge (in) from the multiplexing of multiple sources into a single stream. This declarative construct (1) maps many inputs to the consuming node, and (2) defines the mergestrategy used to produce one ordered output stream.

Code 26: edge.MultiPath(...) example

|  |
| --- |
| edge.MultiPath(  inputs=[  <EdgeSpecifier>,  edge.Pipeline(…),   ],  merge=Sort(  algorithm=ascendingTimeStamp,  window=120,  key=event.Created,  ),  ), |

1. Arguments
   * inputs: ordered list of input EdgeSpecifier identifiers, representing independent event streams. inputs[0] MUST be the default upstream edge (edge.FIFO(...) or edge.LIFO(...)). Additional entries can include edge.Pipeline(name=...) sources.
   * merge: strategy for combining concurrent inputs into one stream. AMI admits multiple strategies; the common case is:

Code 27: Merge clause example

|  |
| --- |
| merge = Sort(  algorithm = ascendingTimeStamp,  window = 120,  key = event.Created,  ) |

This guarantees that, within a 120-second window, events are emitted in ascending event.Created order.

**Throughput & backpressure.** Merging introduces additional work at Collect(). Tune minWorkers/maxWorkers to maintain ingest capacity. When the Collect() node exhausts resources, its upstream edges apply their **own** backpressure policies independently (see §2.2.7.2).

#### <EdgeSpecifier>: edge.Pipeline(…)

edge.Pipeline() is a compiler-generated construct, analogous to edge.FIFO() but **without** a edge.LIFO() variant. It is purely declarative and bridges two pipeline segments, identifying the upstream pipeline by its name (the Ingress() node’s name attribute). It must specify minCapacity, maxCapacity, a backpressure policy, and a compatible type (see §2.2.7).

### Egress Node

Figure 24: Egress Node Topology

A diagram of a system

AI-generated content may be incorrect.

The Egress() node resembles the Transform() node in several ways, as illustrated below:

Code 28: AMI Egress node example

|  |
| --- |
| ).Egress(  in=edge.FIFO(  minCapacity=10,  maxCapacity=20,  backpressure=block,  type=someProject.CsvRecord,  ),  worker=db.ExecStoredProc(  proc="csvRecordInsert",  conn=dbConnection(  dbHost=config.Database.get("dbHost"),  dbPort=config.Database.get("dbPort"),  dbUser=config.Database.get("dbUser"),  dbPass=config.Database.get("dbPass"),  ),  ),   minWorkers=2,  maxWorkers=8,  capabilities=[  NetOut: "db.acme-corp:5432",  ], ) |

But the Egress() node is the terminator for a pipeline and has no downstream edge. It receives Event<T> objects from an upstream edge then uses a worker to deliver the event data as the desired result to some sink. The worker functions are dispatched like they are with Transform() or any other node. A pool of worker virtual machines scales dynamically between minWorkers and maxWorkers. But the workers have less restrictions than other nodes, having I/O facilities like Ingress() nodes.

The I/O capabilities defined by an Egress() node must be specific access controls to ensure least-privilege access to the system beyond the AMI program.

### Egress().get()Configuration Property Streams

Not all pipelines write to the external world. Some terminate at an **Egress()** node whose products are consumed by other pipelines. In the running example, the egress publishes configuration used by a downstream worker:

worker=db.ExecStoredProc(  
 proc="csvRecordInsert",  
 conn=dbConnection(  
 dbHost=config.Database.get("dbHost"),  
 dbPort=config.Database.get("dbPort"),  
 dbName=config.Database.get("dbName"),  
 dbUser=config.Database.get("dbUser"),  
 dbPass=config.Database.get("dbPass"),  
 ),  
)

Here, config.Database.get(<propertyName>) retrieves values from a configuration pipeline. The .get() method exposes key–value pairs whose keys are declared as expose\_<name>; values are returned **by copy**, preserving the node-memory boundary and preventing direct access to another node’s state.

#### Security Implications

Because configuration streams can originate from secure sources (encrypted files, secret stores, or hardware-backed mechanisms), they avoid many pitfalls of static config files. Reads are observable via backpressure release, enabling policies such as secret rotation or access logging.

#### Blocking Behavior

Egress().get() is a **blocking** call: if a consumer requests a value that is not yet available, the call blocks until the configuration pipeline produces one. Program designs should account for this return-once, blocking behavior.

#### Configuration Pipeline Interface Methods

An Egress()node exposes several methods for consumption by other pipelines:

Table 11: Egress Node Methods

|  |  |
| --- | --- |
| Method | Purpose |
| .get(<property\_name>) T | Allows caller to retrieve a corresponding property value |
| .list() []string | Allows a caller to enumerate the list of properties |
| .count() uint | Returns the count of exposed properties. |

### Worker Function Factory Pattern

In the Egress(), db.ExecStoredProc(...) is a factory that returns the worker bound to Egress(). The factory invokes dbConnection(...) to assemble a connection from node-state configuration:

worker=db.ExecStoredProc(  
 proc="csvRecordInsert",  
 conn=dbConnection(  
 dbHost=config.Database.get("dbHost"),  
 dbPort=config.Database.get("dbPort"),  
 dbName=config.Database.get("dbName"),  
 dbUser=config.Database.get("dbUser"),  
 dbPass=config.Database.get("dbPass"),  
 ),  
 ),

The compiler treats the factory as a decorator of the underlying worker. Arguments whose values are statically known (such as the literal proc) are folded at compile time; values that depend on runtime lookups (such as conn, which is derived from config.Database.get(...)) are resolved at runtime when the factory executes and before it returns the worker. If a configuration key is missing, dbConnection(...) must apply a deterministic fallback (for example, a default, an error, or an environment override). Because Ingress() and Egress() sit on the security boundary, they may read environment variables only when explicitly authorized by their capability lists; this permits the Egress factory to complete initialization using environment values when node-state configuration is absent. In contrast, Transform() is pure and does not consult external state; any configuration for a transform must arrive in-band in Event<T> or be fixed at compile time. These rules preserve compile-time type checking, the single-disposition rule, and the capacity/backpressure semantics of the consumer edge.

### Node-State Tables

Direct memory sharing between worker virtual machines is not allowed in AMI or POP. But shared state between workers is allowed through a special node-state table facility. This key-value table provides a mechanism for safely sharing state between workers which are otherwise autonomous.

#### Data Structure

The node-state table exposes a key-value store. Its internal operation is opaque to the programmer and the worker virtual machines. The data structure internally must guarantee fast lookups and atomic operations in memory. This is an ephemeral state table only.

#### Methods

The node-state table exposes the following methods to worker virtual machines and their functions:

Table 12: Node-State Table Methods

|  |  |  |
| --- | --- | --- |
| Method | Return Type | Description |
| .state.get(name string, block bool) (any, error) | any | Returns the value associated with a given property name. |
| .state.set(name string, value any) error | error | Insert/update the value for a given property name. |
| .state.update(name string, value any) error | error | Update an existing property with the given value. |
| .state.delete(name string) error | error | Delete a given property. |
| .state.list(offset, limit uint) []string | []string | Returns a paginated list of properties. |
| .state.count() uint | Uint | Returns the number of properties in the table. |

#### Strong Typing and Node-State

The node-state table is strongly typed. All data stored in the node-state table will be stored as a name (string) and value ([]byte), and the []byte data value will be byte-serialized with a header identifying type and size followed by a series of payload bytes.

#### Byte-Serializing Node-State Records

Data is serialized before being stored to the node-state table by either .set() or .update(). It is deserialized before being returned by .get(). Every data object is serialized with a 64-bit header (32-bit type plus 32-bit size), as follows:

Figure 25: Node-State Record Structure

|  |  |  |
| --- | --- | --- |
| Type | Size | Data Object Bytes |
| 32 bits | 32 bits | Variable-size |

Each data type (built-in and user-defined) is assigned a numeric type for use with the node-state table:

Figure 26: Node State Record Types

|  |  |  |  |
| --- | --- | --- | --- |
| Type | NSR Type Value | Type | NSR Type Value |
| bool | 1 | Float32 | 11 |
| byte | 2 | Float64 | 12 |
| int | 3 | uint | 13 |
| int8 | 4 | uint8 | 14 |
| int16 | 5 | uint16 | 15 |
| int32 | 6 | uint32 | 16 |
| int64 | 7 | uint64 | 17 |
| string | 8 | rune | 18 |
| map | 9 | set | 19 |
| slice | 10 | array | 20 |

All other user-defined types (enum, struct, alias) will be assigned numeric identifiers greater than 1000 by the compiler. There is no need for global uniqueness since the type assignments are local only to the node-state table for a given node.

Node-state tables can support up to 4GB per node-state record given the 32-bit size field. However, larger values in the node-state table should be discouraged as the performance penalty would not be trivial.

#### state.get(name string, block bool) method

The state.get() method returns the value associated with a given property name. However, if the property does not exist, the method will return an error object unless the block value is true. When block is true, the .get() method will wait for the property to exist or the node-state block deadline to expire. The node-state will lock during operations to ensure all workers have safe access to node-state.

#### state.set(name string, value any) method

When a worker node calls .set(), the value is serialized and a lock is requested from the node-state table. When the lock is granted, the serialized value is written. The .set() method will insert new records, or it will update existing records.

If .set() attempts to update a record and the record types do not match, an exception will be thrown. The compiler should work to identify all possible node state properties and guarantee type mismatch does not occur. The runtime should enforce these guarantees as well.

#### state.update(name string, value any) method

Workers call .update() to update existing records. If .update() is called and the property does not exist, an error will be thrown. If the data type of the record to be updated does not match the value to be applied with the update, an error will be thrown. The compiler should work to identify all possible node state properties and guarantee type mismatch does not occur. The runtime should enforce these guarantees as well.

#### state.delete() method

The *.delete()* method removes a property from the node-state table. If the property does not exist, the method will return an error.

#### State.list() and state.count() methods

The .list() and .count() methods are simple ways of enumerating and counting the records in the node-state table at a given time. These will both lock the table for a brief time while they collect their data.

## Imperative Programming in AMI

The previous section described AMI programs at the pipeline level using the Declarative Pipeline Descriptor Language (DPDL). This section introduces the Imperative Worker Function Language (IWFL), which specifies the worker functions that execute within nodes and the reusable functions from which those workers are composed.

### Global Scope, Defined

We begin with three rules:

1. Exactly five kinds of identifiers may be declared at global scope: package, pipelines, datatypes (enum, struct), constants, and functions.
2. There are no global variables in AMI.
3. Globalconstants are allowed and are resolved by the compiler in both the declarative (DPDL) and imperative (IWFL) code.

AMI has no global variables. The only identifiers permitted at global (package) scope are packages, pipelines, data structures (enum, struct), constants, and functions. Constants may appear at global scope and are folded by the compiler wherever possible in both declarative and imperative code, but mutable data is never globally visible.

For imperative code, the effective “global” boundary is the worker’s sandboxed virtual machine. A worker observes exactly one input and may return at most one output and one error, following the uniform signature defined in §2.2.6:

Code 29: Worker Function Signature

|  |
| --- |
| func <identifier> (in Event<N>) (out Event<T>, err Error<E>) |

The worker VM exposes no ambient state beyond this contract. Any state that must persist across invocations is confined to the node-state table, which is addressable only through the API described later in §2.3.5 (state.get(), state.set(), state.update(), state.list()). Inputs are immutable copies presented to the worker; outputs are immutable Event<T> values (or a typed Error<E>) returned to the runtime, which performs the downstream routing and error dispatch according to the pipeline graph. This separation ensures that imperative code remains locally reasoned, while lifecycle, ordering, and backpressure remain declarative properties of the pipeline (§2.2.7, §1.1.7).

These rules make IWFL predictable to analyze and safe to optimize. The compiler can specialize each worker VM to its declared capabilities and types (§2.2.5; §2.2.12), while the runtime enforces the single-disposition rule (§2.2.6.3): on each invocation a worker either emits one Event<T>, emits no event (“silence”), or returns an Error<E>, but never more than one of these outcomes. Together, these constraints allow AMI to combine high-level declarative structure with precisely bounded imperative behavior.

### Pointers and Addresses

AMI does not expose raw pointers or process addresses. The \* token is **not** a dereference; when used on the **left-hand side** it explicitly marks a mutating assignment.

Code 30: Mutation operator (\*) demonstrated

|  |
| --- |
| var foo int // local bindings are immutable by default  \*foo = 3 // explicit in-place update |

Without \*, the assignment is immutable. The \* marker must obey all rules of the mutate operator (§2.4).

#### Passing by Reference Not Allowed

AMI forbids **pass-by-reference**. All values are immutable. Function parameters are passed **by value**, and results are returned **by value**. No function can observe or mutate data outside its own scope.

#### Unused Returns and Arguments

The compiler rejects bindings that introduce an unused name. When a returned value is intentionally discarded, the **underscore identifier** \_ expresses that intent and suppresses the error.

Code 31: Unused function return

|  |
| --- |
| func foo() int {  \_, b:=func()int, int {  //do something  Return 1,2  }()  Return b  } |

Here the first result is explicitly discarded. Because the compiler can see the discard, it may elide the copy of that value entirely.

The same convention applies to parameters:

Code 32: Unused function parameter

|  |
| --- |
| func foo(\_ int, b int) int{  return b  } |

Using \_ at the call site is permitted only when the callee does not read that parameter. If the parameter is referenced, the program fails to compile:

Code 33: Illegal omitted input parameter

|  |
| --- |
| func foo(a int, b int) int{  return a + b  }  foo(\_,2) |

If the callee ignores a parameter, the discard at the call site is valid:

Code 34: Allowed input parameter

|  |
| --- |
| func foo(a int, b int) int{  return b  }  foo(\_,2) // allowed; the first argument is intentionally discarded |

In all cases, \_ denotes a value that is computed but not bound to a usable name. The compiler treats it as an explicit instruction to throw the value away and may optimize away the associated copy or allocation.

### Function Declaration Forms

IWFL admits two ways to define computation:

* + **named functions** are declared at package scope and anonymousfunctionliterals that appear as expressions. Both forms obey the same typing rules and multiple-return semantics, and both are compiled under the same capability constraints as the node that calls them. Named functions provide reusable building blocks;
  + **anonymous functions** are convenient for short, localized computations and immediate invocation.

#### Worker Functions

A workerfunction is a special case of a named function whose signature conforms to the uniform worker contract introduced in §2.2.6:

Code 35: Worker Function Signature

|  |
| --- |
| func <identifier>(in Event<N>) (out Event<T>, err Error<E>) |

#### Helper Functions

A **helper function** is any named function with ordinary parameters and results:

Code 36: Example helper function

|  |
| --- |
| func ParseHeader(line string) (Header, Error<E>){  . . .  } |

Helper functions called from Transform() workers must be pure with respect to external state; calls from Ingress() and Egress() must honor the node capabilities. The compiler enforces these constraints when shaping the worker’s VM.

#### Anonymous Function

An anonymousfunctionliteral may be used where an expression is expected and can capture local variables:

Code 37: Anonymous function example

|  |
| --- |
| result := func(x int, y int) (int, Error<E>) {  // …computation…  return x + y, nil  }(a, b) |

Anonymous functions can also be bound to variables and passed as values. The blank identifier \_ applies equally to unused return values and parameters.

#### Factory Function

A factoryfunction constructs a worker by returning a function that matches the worker signature. The factory may capture configuration and is eligible for constant folding and inlining when its arguments are compile-time constants:

Code 38: Factory function pattern

|  |
| --- |
| func MakeInserter(conn DBConn, proc string){  return func(in Event<N>) (Event<T>, Error<E>) {  // …returns a worker that performs the insertion…  }  } |

Functions are declared only at package scope; there is no overloading by arity or parameter type, and names must be unique within a package. Nested **named** declarations are not permitted, but nested **anonymous** functions are. All arguments and results are passed and returned by value (no pass-by-reference; §2.4.2).

#### Generic Form

The generic form of a function permits type abstraction for advanced use cases and increases code reuse. Generic functions **are allowed** as helpers but arenotpermitted as worker functions in a pipeline; workers must declare concrete N, T, and E types to preserve analyzability and static planning. Generic helpers are monomorphized at call sites and remain subject to the same pass-by-value and immutability rules.

**Generic syntax rule.** AMI uses angle brackets <> for all generic declarations and applications. Square brackets [] are reserved for indexing, slicing, and collection literals.

Code 39: Generic Function Form

|  |
| --- |
| func Map<T any, U any>(xs []T, f func(T) U) []U {  out := make([]U, len(xs))  for i, v := range xs {  out[i] = f(v)  }  return out  } |

This Map function abstracts the input and result element types T and U. It applies f to each element of xs and returns a new slice with the transformed values.

#### Method Form

Functions may be declared as **methods** on a struct type. Because struct fields are immutable, a method does not mutate its receiver in place; instead, it computes and returns a new value.

Code 40: Struct method form

|  |
| --- |
| struct Foo {  E int  I int  }  // Returns an updated Foo; the receiver itself is not mutated.  func (a Foo) With(n, m int) Foo {  var b Foo = a // local copy  \*b.E = n // explicit in-place update of the local copy  \*b.I = m  return b // caller receives the new value  } |

This pattern preserves AMI’s copy-in/copy-out semantics: the receiver a is passed by value, any local updates use the explicit mutation operator \* on local bindings, and the updated struct is returned to the caller.

### Data Mutability

#### Immutability at Pipeline Layer

At the pipeline’s declarative layer, all memory is immutable. Mutation is permitted only inside functions, and only where explicitly marked.

#### Local, Explicit Mutation

AMI IWFL is immutable by default. A local binding may be updated in place only when the left-hand side is prefixed with the mutation operator \*. Without \*, reassignment is rejected at compile time.

Code 41: Mutability Example using \* operator

|  |
| --- |
| // immutable computation: create a new binding  var a int = 1  var a2 int = a + 1  // explicit in-place update of an existing local  \*a = a + 1 |

#### Mutability Operator (\*)

The \* marker applies only to eligible local bindings. Event<T> payloads, constants, and node-state are **never mutated directly**. Each \*-modified assignment is atomic and confined to the function’s VM so that effects cannot escape the invocation. This operator follows all mutability rules (§2.3.4.6).

#### Node-State and Mutability

Node-state table changes occur only through state.set(...) or state.update(...), and these operations are atomic.

#### Event<T> is Never Mutable

Events are never mutable and only produced only by returning Event<T>:

Code 42: Example returning Event<T>

|  |
| --- |
| out:=func() Event<int> {  return Event<int>(1)  }()  payload:=out.payload //payload would have a value 1.  created:=out.metadata[“created”] |

In the above example, we see that the Event<T> constructor handles metadata initialization. That metadata is read-only by the programmer and managed behind the scenes by the event lifecycle.

When Event<T> is passed as a function argument, the event object is passed by value into the function, and the function may access the payload and metadata properties.

Code 43: Immutable Event<T> example

|  |
| --- |
| n:=4  evt:=Event<int64>(n)  out:=func(e Event<int64>) Event<int> {  e.payload=int64(e.metadata[“created”])  return e  }()  payload:=out.payload //payload would have a value 1.  created:=out.metadata[“created”] |

In this contrived example, the event payload is updated and the result is returned to the caller. This operation cannot be made mutable. The following would result in a compile-time error:

Code 44: Illegal Event<T> mutation example

|  |
| --- |
| n:=4  evt:=Event<int64>(n)  out:=func(e Event<int64>) Event<int> {  \*e.payload=int64(e.metadata[“created”]) //This would fail!  return e  }()  payload:=out.payload //payload would have a value 1. |

#### Mutate Operator

The mutate operator is the long-form of the \* operator; the syntactical difference is intended to more clearly wrap complex operations where the result is mutable and atomic:

Code 45: mutate operator example

|  |
| --- |
| var result int  result=mutate(funcA(x)+funcB(y)+funcC(z)) |

The compiler will interpret mutate(<expression>) as an atomic operation on the expression. This may allow a wider array of optimizations (including mutability) focused on performance. The mutate operation must follow all of the mutability rules (§2.3.4.6).

#### Mutation Rules

Any mutation operation *must* follow these rules:

1. The compiler guarantees that the operation is atomic:
   * + For \* operators, the assignment is atomic:

Code 46: Atomic mutate operator (\*)

|  |
| --- |
| var result int  \*result=funcA(x)+funcB(y)+funcC(z) |

This means the right-hand side is evaluated, then the assignment is performed as a mutable atomic operation.

* + - For mutate(…) the *expression* is atomic:

Code 47: Atomic mutate expression

|  |
| --- |
| var result int  result=mutate(funcA(x)+funcB(y)+funcC(z)) |

The mutate(…) operator makes the expression *and* assignment as an atomic operation. This gives the compiler significant latitude to optimize the expression and assignment under the guarantee that the entire operation is a safe, protected unit.

1. A mutable operation must exist within the same scope.

Code 48: Mutable operation scope

|  |
| --- |
| {  var result int  result=0  func(){  \*result=5 //mutation allowed in a child scope.  }()  \*result=3  } |

Mutation can occur in a child scope, including a nested *anonymous* function. This is predictable and analyzable because there is no concurrency inside the worker function scope.

1. The result of an operation must be the same type and byte size as the assignee.

Code 49: Size-match mutability rule example

|  |
| --- |
| a:=[]byte{01, 02, 03, 04} // initialize ‘a’ with 4 bytes  b:=make([]byte, 4) // zero-initialize ‘b’ with 4 bytes  c:=make([]byte, 10) // zero-initialize ‘c’ with 10 bytes  d:=make([]byte, 3) // zero-initialize ‘c’ with 10 bytes  \*b=a // mutability is allowed  \*c=a // mutability will fail  \*d=a // mutability will fail  \*a=[]byte{05, 06, 07, 08} // mutability will succeed. |

1. Event<T> payloads, constants, and node-state values are **never** mutated directly; node-state is changed only via state.set() or state.update() and events are produced only by returning Event<T>.
2. Mutability cannot traverse the worker virtual machine boundary (no shared memory across VMs).
3. Except as otherwise provided here, mutability cannot traverse a function boundary.
4. The compiler verifies that mutability follows these rules, and the runtime enforces the same rule set.

### Function Access to Node-State Table

A function may read and write the node’s shared state through the reserved identifier state and its compiler-generated methods: .get(...), .set(...), .update(...), and .list(...). The node-state table and these methods are synthesized by the compiler; state is a special object, not an ordinary value. It cannot be bound to a variable, passed as an argument, or returned from a function. It is visible from any function scope within the node’s VM. Calls to state operate on the node’s state, not on copies: reads return values bycopy; writes create new values and commit them atomically. State never crosses node boundaries; when information must be shared with other pipelines, it is exposed explicitly through programmatic interfaces such as Egress().get() rather than by direct memory access.

### Concurrency

#### Concurrency In Pipelines

Pipelines are concurrent by construction: each node runs in its own worker VM pool, and the controller for a node dispatches invocations without blocking other events (§2.2.6). Backpressure, capacity, and ordering are properties of edges (§2.2.7); node-level concurrency never circumvents those constraints.

#### Concurrency In Functions

Functions execute inside a worker’s VM. AMI does not provide an explicit concurrency keyword (there is no analog to Go’s go). Instead, the compiler may introduce **local parallelism** within a function when doing so does not change its observable behavior. The result must be identical to the sequential evaluation of the same expression.

Consider this example:

Code 50: Example function

|  |
| --- |
| func foo(a, b, c int) int {  return firstFunc(a) + secondFunc(a, b, c) + thirdFunc(c)  } |

A sequential implementation evaluates the three calls in source order and returns their sum. If the compiler can prove that the calls are **independent**—that they read only their arguments, do not access state.\*, do not perform I/O (or, in the case of Ingress() and Egress(), perform only capability-authorized I/O that is semantically independent), and do not rely on evaluation order—then it may evaluate the calls on separate cores and combine the results. The worker VM still presents a single invocation boundary: all local work must complete before the function returns, preserving the single-disposition rule and event-level atomicity.

The following constraints govern such optimization:

* + **Determinism**.

Parallel evaluation must be observationally equivalent to the sequential program: same return values, same state.\* effects (if any), and the same emissions on success or failure. When evaluation order is semantically significant, the compiler must preserve it.

* + **Isolation**.

No shared memory exists between VMs, and within a VM the compiler must avoid introducing data races. Local temporaries may be used freely; inputs and outputs remain copy-in/copy-out.

* + **Capabilities.**

For Transform() workers, only pure helper calls are candidates for parallel evaluation. For Ingress() and Egress(), helper calls that exercise capabilities may run in parallel only when their effects are independent by construction (e.g., disjoint files or distinct network endpoints) and the capability policy permits it.

* + **Cost model.**

The compiler applies this optimization opportunistically and only when the expected latency or throughput benefit exceeds the scheduling overhead; the semantic contract takes precedence over speed.

Under these rules, a function like foo may be parallelized by the compiler, but a function that touches node state, relies on timing, or performs ordered I/O will run sequentially.

### Enumerated Types (enum)

An enum defines a finite set of named constants. By default, the underlying type is int, starting at 0 and incrementing by 1. Explicit values are allowed; explicit sequences must be monotonic (ascending or descending) but need not be contiguous.

Code 51: Enum example

|  |
| --- |
| Enum Colors {  Red,  White,  Blue  }  Enum Buckets {  A = 1,  B = 2,  C = 3  }  Enum Evens {  Two = 2,  Four = 4,  Six = 6,  }  Enum OddsDown {  Seven = 7,  Five = 5,  Three = 3,  } |

#### Arithmetic.

If the values are contiguous, the compiler permits integer arithmetic with modular wrap within the enum’s range. If they are non-contiguous, arithmetic is disallowed and will result in a compile-time error; comparisons and equality remain valid.

#### Typing and scope.

Enum names are types; members are constants scoped to the enum’s namespace. Enums are immutable and have compiler-provided trivial destructors (zeroize + free). This aligns with AMI’s ownership/RAII rules.

#### .String() Method

For ergonomic use, the compiler synthesizes a pure .string() method for every enum type. This method returns the exact spelling and case of the constant as declared—locale-independent and stable across builds—and runs in constant time by consulting a compiler-emitted lookup table. It allocates only the returned string. The method is defined for all valid enumeration values.

Code 52: Enum .String() example

|  |
| --- |
| enum Colors { Red, White, Blue }  var c Colors = Colors.White  var s String = c.String() // "White" |

### Struct Types (struct)

A struct is a nominal type composed of named fields, each with a declared type, where the field ordering describes the in-memory field layout of the runtime data structure:

Code 53: struct example

|  |
| --- |
| struct Point {  X Int  Y Int  } |

#### struct Immutability

A struct literal constructs a value; by default, that value is immutable. Ordinary field assignment is permitted but is **pure**: the expression p.X = 5 does not alter p in place. Instead, the compiler constructs a new Point whose X is 5 and whose remaining fields are copied from p, and then **atomically** replaces the binding on the left-hand side with that new value. If the allocation required for the replacement cannot be satisfied within the VM’s budget, the replacement does not occur and the binding remains unchanged. This copy-on-update semantics applies uniformly to all field types, including dynamically sized fields (e.g., String, Map, Set, Slice), and requires no special syntax.

#### struct Mutability

AMI also provides an explicit opt-in for in-place updates when it is safe to do so. The mutation operator applies to the **left-hand side** and may target either an entire local binding or a field of a local binding:

* + \*p = Point{ X: p.X + 1, Y: p.Y } replaces the local binding p in place.
  + \*p.X = 5 updates the field X in place.

Both forms are permitted only when the enclosing struct is a **local-scope binding** and the general mutation rules are satisfied (§2.3.4.7). In particular, the VM may perform the operation in place **only if** the new value’s size does not exceed the **allocated size** of the target. For fixed-width scalars this condition is trivially true; for dynamically sized fields (e.g., String, Map, Set, Slice) it may fail. When it fails—or when the compiler cannot prove it will hold—the runtime **atomically** converts the operation to the immutable path described above (construct a fresh struct, then replace the binding). If allocation still cannot be satisfied by the allocator before its deadline, the replacement does not occur and the binding remains unchanged (§2.4.3.3).

The operator is rejected when the target is not a local binding (for example, an event payload, a value obtained from state.get, or a captured value that the compiler cannot prove local to the VM). Mutation never crosses VM boundaries and never modifies node-state directly; changes to node-state occur only through state.set or state.update (§2.2.14).

#### Struct Method Mutability

A struct method is an atomic operation, and the method may mutate struct fields as they are local to the method, provided the struct fields satisfy the mutability rules (§2.3.4.7).

### Dynamic Containers (map, set, slices)

Dynamic containers are ordinary values. To the developer they can be instantiated as follows:

Code 54: Map, set, slice allocation

|  |
| --- |
| m:=make(map<int, int>) // allocate a map (empty)  s:=make(set<int>) // allocate a set (empty)  slice:=make([]byte,4) // allocate a slice of 4 bytes |

#### Internal Container Representation

Internally, the compiler represents these as a container object and an immutable data element:

Figure 27: Dynamic memory container topology

A diagram of a data flow

AI-generated content may be incorrect.

The container is represented by a single memory pointer, which the object code uses to reference a size-prefixed data object. The container pointer is updated to a new data object reference when immutable operations occur on the data object. Each data object and its prefix occupy a contiguous memory region.

#### Container Data Operations (Mutable and Immutable)

When data is passed from one function to the next, the container is recreated in the new local scope and a data object of equal size is created, to which the original data object is copied. The copy is owned by the called function, and when the function terminates, this container copy is destroyed consistent with RAII.

While inside a local scope, the owner of a container may mutate the state of a container value, provided this would not result in a change in data object size. But if a state change would result in a larger or shorter data object size, then the container operation is immutable. The container is locked, a new data object is provisioned, and the state of the original data object is transferred into the new data object. Then the new state change is applied.

#### Empty Containers

When a container is declared or instantiated, it is an empty (or nil) container:

Code 55: Empty containers

|  |
| --- |
| var emptyMap map<int, int>  var emptySet set<int>  var emptySlice []byte  m:=make(map<int, int>) // allocate a map (empty)  s:=make(set<int>) // allocate a set (empty)  slice:=make([]byte) // allocate a slice of 4 bytes |

It contains no values; thus, the following conditions would be true:

Code 56: nil-container example

|  |
| --- |
| if (m == nil) && (s==nil) && (slice==nil) {  // this is all true  } |

This initial internal state may be visualized as follows:

Figure 28: Empty Container

A diagram of a computer

AI-generated content may be incorrect.

When data values are assigned to a container in this empty state, a new size-prefixed data object is provisioned and the container’s pointer is updated to reference this new data object.

#### Strings as a Container

A string is a special type of container. A string has no nil value as its empty state. Its empty state is simply an empty string, represented as "". Nonetheless, the string is always immutable. Further, a string’s internal state is never mutated. Any operation on a string will result in an immutable operation (i.e. the provisioning of a new string).

#### Dynamic Memory in worker VMs

Any dynamic memory provisioned by a worker function (or its child functions) is provisioned from the heap of the worker virtual machine hosting the function. When the function terminates, the memory is reclaimed to that worker VM’s heap.

No direct memory sharing is allowed between virtual machines.

If a worker function requests dynamic memory from its worker virtual machine heap which exceeds the virtual machine heap, a memory allocation error will occur. The function may handle this error or allow it to bubble up to the worker function which may return an Error<E> to the worker virtual machine. This Error<E> object would then be routed to the appropriate error handling pipeline.

#### Dynamic Memory and Node-State Tables

Any dynamic memory container state (e.g., slices, map, set, string) which is shared across virtual machines via the node-state table facility is first copied into the node-state table using .set() or .update() then loaded into the target virtual machine(s) using .get(). The data capture by .set() and .update() is atomic and consists of a byte-serialized data representation of the container, its type, size and data values (§2.2.14).

## Memory Management

AMI guarantees RAII-style memory management: objects are created with a clear owner and deterministically released when their owner’s lifetime ends. Ownership is simple:

Code 57: Allocation domains (abstract)

|  |  |  |
| --- | --- | --- |
| Domain | Controller | Description |
| Pipeline | compiler | Edge buffers are declarative, sized and bounded at compile time; backpressure policies govern run-time growth/limits (§2.2.7). |
| Worker | runtime | Each worker runs in an isolated VM with its own heap/arena; there is no shared memory across VMs (§2.3, §2.2.6.5). |
| Events | runtime | Functions are pure by default. Local, explicit mutation is opt-in via the \* (mutable) operator on the left-hand side (§2.3.4). |
| Node-State | runtime | Memory is separate, transactional, and accessed only via state.get/set/update/list (§2.3.4.4). |

### Allocation Domains

Figure 29: Memory Allocation Domains

A diagram of a company's diagram

AI-generated content may be incorrect.

AMI partitions memory into a small number of analyzable allocation domains. Each domain has a clear owner, well-defined lifetime boundaries, and explicit rules for growth and release. This organization enables RAII-style management across the pipeline and worker layers while preserving POP’s determinism and observability guarantees.

#### **Edges (pipeline layer).**

Edge storage is sized in objects, not bytes: , where is the edge capacity (bounded by its declared min/max) and is the size of the concrete Event<T> carried on that edge. Capacity may grow at runtime from minCapacity up to maxCapacity; once maxCapacity is reached, the declared backpressure policy governs behavior. Because types—and thus —are known at compile time, these bounds are both statically checkable and auditable.

In steady state (“happy path”), occupancy fluctuates within the declared bounds, and no items are dropped. Under pressure (“sad path”), policies such as block, dropNewest, or dropOldest apply deterministically; the system may emit an Error<E> for observability while maintaining analyzable behavior.

#### **Worker VM heap (imperative layer).**

Each worker invocation runs inside an isolated VM with its own heap/arena. All per-call allocations—locals, temporaries, and dynamic containers (map, set, slices)—live in this domain. When the worker returns (success, silence, or error), the VM releases and zeroes its arena, bounding the lifetime of all such memory to the call. No shared memory exists across VMs; inputs and outputs are copy-in/copy-out, consistent with AMI’s immutability model and uniform worker signature.

If a requested growth (e.g., container resize) would exceed the VM’s budget or deadline, the allocation fails atomically: the worker returns err=Error<E> and no out event; any in-flight state.update() is rolled back and locks are released. These failures inform the scheduler’s error-aware policies without violating event-level atomicity.

#### **Node-state store (shared, key-scoped).**

Node-local state is managed in a separate store with key-level atomicity and rollback semantics. It persists across worker invocations but is accessible only through the state.get/set/update/list/count APIs; it is never mutated by direct memory access and is scoped to the node. This separation keeps per-call VM memory analyzable while allowing controlled, transactional sharing where needed.

#### **Compiler data (read-only).**

Constant data and generated code reside in read-only segments and are never mutated at runtime, preserving referential transparency for analyses and optimizations.

#### Ownership and transfer.

Objects are owned by the scope that creates them. Returning a value or enqueuing an Event<T> transfers ownership to the receiver (caller or downstream edge). Destruction follows scope: temporaries die at scope end; a worker’s heap is reclaimed at VM teardown; edge elements are released when removed or when capacity drops under policy. This simple model underpins RAII across domains.

**Examples**

***Happy path (per-call lifetime and transfer)****:*  
A worker constructs locals, produces a new event, and returns; locals die with the VM, and ownership of the event transfers to the downstream edge.

Code 58: Per-call lifetime and transfer example (happy)

|  |
| --- |
| func Scale(in Event<Image>) (Event<Image>, Error<E>) {  var tmp = Convolve(in.payload, KernelSharp3) // local allocs in VM  return Event<Image>{payload: tmp}, nil // transfer to edge  } |

**Sad path (container blow-up caught by the VM budget):**  
Unbounded inserts cause the VM’s allocation budget to be exceeded; the operation rolls back atomically, and the worker returns an error with no emission.

Code 59: Per-call lifetime and transfer example (sad)

|  |
| --- |
| func Explode(in Event<Blob>) (Event<Blob>, Error<E>) {  var m map<string>Blob  for x in in.payload.items {  \*m = put(m, x.key, x.value) // growth may exceed VM budget  }  // On budget exceed: atomic rollback, return nil + Error<E>  return Event<Blob>{payload: m}, nil  } |

Together, these domains—edges, worker VM heap, node-state, and compiler data—provide tight, analyzable bounds on allocation, growth, transfer, and destruction, aligning memory behavior with POP’s static graph, immutable events, and capability-gated effects.

### Ownership & RAII

AMI enforces **single ownership** for all VM-resident objects. At any moment exactly one scope owns a value; ownership may be **moved** into a child scope and may be **returned** to the parent via a function result, but it is never duplicated or jointly held. Copies create independent storage and never imply shared writable memory. Under these constraints the compiler can plan lifetimes ahead of time and the runtime reclaims deterministically at scope exit.

#### Node-State Tables and Immutable Events

Node-state values are excluded from the ownership regime described in this subsection: the table owns its values; reads return copies into a VM, and writes install new values atomically through the state APIs.

Events are immutable; no binding ever “owns” the same event as another binding across a VM boundary.

#### Ownership Attaches to Bindings

Inside a worker VM, a binding owns the values it creates (scalars, structs, and dynamic containers such as maps, sets, and slices). Passing arguments copies values into the callee’s VM; returning values transfers ownership back to the caller. Emitting an Event<T> transfers ownership to the downstream edge; dequeuing transfers ownership to the consumer VM.

#### What It Means to “Own”

A binding that owns an object controls its lifetime. A reassignment **without** the \* (mutable) operator constructs a **new** value; the old value remains owned by the prior binding until that binding leaves scope. A reassignment **with** \* performs an **in-place** update of that binding, consistent with the mutability rules; any internal resize or rehash is carried out atomically by the VM allocator, and no partial state becomes visible.

#### Directional Transfer of Ownership

Ownership moves in clear, **one-direction** steps:

* + **Caller → callee** (argument passing by value).
  + **Callee → caller** (return by value).
  + **Worker → edge** (returning Event<T>).
  + **Edge → consumer VM** (delivery).

There is no lend/borrow that resumes ownership implicitly. If a callee must retain a value, the caller must return it; otherwise, the callee’s copy dies at scope end.

#### Finalization at Scope Exit

When a binding falls out of scope, AMI runs destruction in **last-in/first-out** (LIFO) order for that scope, on both success and error paths. For VM-resident objects, destruction means: (a) the program relinquishes ownership; and (b) the VM ensures the storage is securely reclaimed. **Capability-backed handles** (files, sockets, DB connections) are released via compiler-generated wrapper code that calls the handle’s explicit close/release API before the VM frees its heap; destructors themselves never perform side effects.

#### Destructor Semantics and Asynchronous Reclamation

All types have **compiler-provided** constructors/destructors. For trivial types (scalars, enums, strings, fixed-size structs) destructors are trivial. For dynamic containers (map, set, slice) destructors walk internal buffers as needed. In all cases destructors are **effect-free**: they may only enable secure memory cleanup; they do **not** perform I/O, consult capabilities, touch node-state, or block user execution.

Destruction follows a **free-to-fence** model:

* + A destructor **releases** the object to the worker VM’s allocator, transferring ownership to the allocator and rendering the object **unreachable** to the program.
  + The allocator is the sole agent that **zeroizes and frees** storage. It may complete reclamation **synchronously** (before the next allocation proceeds) or **asynchronously** in a short epilogue task. Asynchronous work is registered with the allocator, remains fenced (unobservable), and must complete no later than VM teardown.
  + If a new allocation would exceed the VM budget, the allocator **pauses**, flushes pending epilogues up to the allocation deadline, and then either (i) services the request (budget credited) or (ii) fails the request with a typed allocation error. No partial allocations are installed.
  + Memory is **zeroized before reuse** in any domain (same VM or OS). The two modes are semantically identical: once released, a value is unusable; the timing of zeroization is allocator-internal and never changes program behavior.

#### Secure By Default Memory Design

AMI implements a secure by default memory design. All memory is zeroized on release and before reallocation. The compiler may implement randomized overwrites in the future as an optional feature to further secure data geometry.

#### Failure, Budgets, and Inviolate Ownership

If a requested allocation or growth would exceed the VM’s memory budget or violate a deadline, the allocator fails the operation atomically. No partial object is installed, no ownership changes hands, and no downstream event is emitted. The worker returns a typed error; any active node-state update is rolled back; and scope exit still runs the same destruction sequence. The single-owner rule remains intact across all error paths. With inviolate single ownership, directional transfers only, and compiler-provided, effect-free destructors, AMI’s RAII strategy remains simple to analyze and uniform to enforce. Every VM-resident object has one owner; moves are explicit and acyclic; clean-up is deterministic and free of side effects; and capability lifetimes are coordinated explicitly by the wrapper rather than by user code hidden in destructors.

### Per-VM Memory Management

Each worker invocation executes inside an isolated workervirtualmachine(VM). Inputs arrive byvalue; outputs (Event<T>) and diagnostics (Error<E>) depart byvalue. No writable memory is shared across VMs.

#### Min/Max Workers (Worker VM Pool)

Nodes define minWorkers and maxWorkers to ensure the lower and upper limit on a worker pool for horizontal scaling as load changes. The number of provisioned workers will increase to maxWorkers and if load continues to increase a backpressure event will occur. When new worker virtual machines are provisioned the delay to provision a new worker virtual machine (cold-start) can harm performance. To reduce this cold-start penalty, the minWorkers threshold which guarantees there is minimum capacity.

#### VM Memory Management Entry and Exit

A VM begins an invocation when the controller dispatches an Event<N>. On return (success, silence, or error), the VM:

* + releases all VM-resident allocations via the free-to-fence model (§2.4.2.4),
  + registers any pending zeroization with the allocator (asynchronous reclamation),
  + closes capability handles opened by the call (via compiler-generated wrapper), and
  + resets execution context.
  + awaits the next worker function invocation.

#### Allocation and mutation

Bindings are immutable by default. In-place updates (\* or mutate()) are permitted only under §2.3.4.7. If the new value’s size exceeds the **allocated size** of the existing value/container, the runtime **atomically** reverts to an immutable update (copy-on-grow) to avoid a fatal error (§2.4.4).

#### Allocator deadlines and pressure

If a requested allocation would exceed the VM’s budget, the allocator pauses the request, flushes pending zeroization up to the allocation deadline, then either services the request (budget credited) or fails it atomically with a typed error. No partial allocations are installed; event-level atomicity is preserved.

#### Node-state and closures

Node-state methods (.get(), .set(), .update(), .list(), and .count()) operate in a separate domain; reads copy into the VM, writes replace atomically. Closures capture by value, never escape the VM, and follow the same lifetime as other VM-local objects.

#### Worker Virtual Machine Lifecycle

Figure 30: Worker VM Lifecycle

A diagram of a program

AI-generated content may be incorrect.

A node maintains a pool of worker VMs sized between minWorkers and maxWorkers.

**Warm pool.** The controller keeps at least minWorkers VMs **warm**: the worker function is loaded and the VM is idle, awaiting dispatch. Warm VMs eliminate cold-start latency by avoiding provisioning on the critical path. When an Event<N> arrives, the controller assigns it to an idle warm VM; if none are available and load warrants, additional VMs are provisioned elastically up to maxWorkers.

**Invocation.** The assigned VM runs the worker function. On termination:

* + - any Event<T> is enqueued to the downstream edge and any Error<E> is routed to the error pipeline,
    - VM-resident memory is released to the allocator (free-to-fence; zeroize-before-reuse),
    - capability handles opened during the call are released by the wrapper, and
    - the VM’s execution context is reset.

**Recycle vs. destroy.** If current load does not justify retention, the pool is downsized and the VM is destroyed. Otherwise, the VM is **recycled** to the warm pool. A recycled VM is **observationally identical** to a cold VM:

* + no user memory persists across invocations,
  + all program-visible storage has been released and is unreachable,
  + any deferred zeroization is allocator-internal and fenced,
  + no node-state locks or in-flight updates survive the call, and
  + no capability handle remains open.

At no point may an invocation observe memory or effects from a prior invocation; warm pooling changes latency only, not semantics.

# AMI Toolchain

The AMI CLI (ami) is a single, deterministic, scriptable command-line application for project/workspace management, package operations, linting, testing, and building. All commands are non-interactive by default and support machine-parsable output via --json. The toolchain is implemented in Go (Golang) 1.25+.

Table 13: AMI toolchain

|  |  |
| --- | --- |
| Command | Description |
| ami init | Initialize the AMI workspace. This creates ami.workspace file, etc. |
| ami clean | Delete the ./build directory and recreate it. |
| ami mod clean | Remove and recreate ~/.ami/pkg |
| ami mod update | Update all project dependencies. |
| ami mod get <url> | Download a package to the local .ami/pkg/ directory. |
| ami mod list | List all packages in the local cache |
| ami lint ./… | Lint the project. |
| ami test ./… | Execute every \*\_test.ami test file. |
| ami build | Compile the ami.workspace project. |

Table 14: ami exit codes

|  |  |
| --- | --- |
| Code | Identifier |
| 0 | SUCCESS |
| 1 | USER\_ERROR |
| 2 | SYSTEM\_IO\_ERROR |
| 3 | INTEGRITY\_VIOLATION\_ERROR |
| 4 | NETWORK\_REGISTRY\_ERROR |

Table 15: ami global flags (all subcommands)

|  |  |
| --- | --- |
| Flag | Description |
| --help | Alias for ‘ami help’ |
| --json | Print output as JSON |
| --verbose | Write verbose output |
| --color | Use colored output |

Code Samples

[Code 1: Prolog Marriage Problem Fact 23](#_Toc209623766)

[Code 2: Prolog Marriage Problem Query 23](#_Toc209623767)

[Code 3: Prolog Marriage Problem Infinite Loop 24](#_Toc209623768)

[Code 4: AMI Comment Examples 207](#_Toc209623769)

[Code 5: AMI source package statement example 209](#_Toc209623770)

[Code 6: package versioning rules 209](#_Toc209623771)

[Code 7: Package SemVer Regular Expression 210](#_Toc209623772)

[Code 8: Package definition and imports example 210](#_Toc209623773)

[Code 9: Pragma Pattern 211](#_Toc209623774)

[Code 10: Example ami.workspace 212](#_Toc209623775)

[Code 11: ami.workspace toolchain.\* 214](#_Toc209623776)

[Code 12: Example AMI Source 216](#_Toc209623777)

[Code 13: AMI Package declaration pattern 220](#_Toc209623778)

[Code 14: Example AMI Package Declaration 220](#_Toc209623779)

[Code 15: AMI Import Pattern 221](#_Toc209623780)

[Code 16: AMI Import Examples 222](#_Toc209623781)

[Code 17: AMI Ingress Node Example 224](#_Toc209623782)

[Code 18: Ingress/Egress capabilities clause example 227](#_Toc209623783)

[Code 19: AMI Worker Function Signature 228](#_Toc209623784)

[Code 20: Ingress node 'in' attribute example 231](#_Toc209623785)

[Code 21: AMI Transform Node Example 234](#_Toc209623786)

[Code 22: AMI FanOut node example 236](#_Toc209623787)

[Code 23: Minimal FanOut Schematic 238](#_Toc209623788)

[Code 24: FanOut 'out' attribute example 239](#_Toc209623789)

[Code 25: AMI Collect node example 241](#_Toc209623790)

[Code 26: edge.MultiPath(...) example 243](#_Toc209623791)

[Code 27: Merge clause example 244](#_Toc209623792)

[Code 28: AMI Egress node example 245](#_Toc209623793)

[Code 29: Worker Function Signature 254](#_Toc209623794)

[Code 30: Mutation operator (\*) demonstrated 255](#_Toc209623795)

[Code 31: Unused function return 256](#_Toc209623796)

[Code 32: Unused function parameter 256](#_Toc209623797)

[Code 33: Illegal omitted input parameter 256](#_Toc209623798)

[Code 34: Allowed input parameter 257](#_Toc209623799)

[Code 35: Worker Function Signature 258](#_Toc209623800)

[Code 36: Example helper function 258](#_Toc209623801)

[Code 37: Anonymous function example 258](#_Toc209623802)

[Code 38: Factory function pattern 259](#_Toc209623803)

[Code 39: Generic Function Form 260](#_Toc209623804)

[Code 40: Struct method form 260](#_Toc209623805)

[Code 41: Mutability Example using \* operator 261](#_Toc209623806)

[Code 42: Example returning Event<T> 262](#_Toc209623807)

[Code 43: Immutable Event<T> example 263](#_Toc209623808)

[Code 44: Illegal Event<T> mutation example 263](#_Toc209623809)

[Code 45: mutate operator example 264](#_Toc209623810)

[Code 46: Atomic mutate operator (\*) 264](#_Toc209623811)

[Code 47: Atomic mutate expression 265](#_Toc209623812)

[Code 48: Mutable operation scope 265](#_Toc209623813)

[Code 49: Size-match mutability rule example 265](#_Toc209623814)

[Code 50: Example function 268](#_Toc209623815)

[Code 51: Enum example 270](#_Toc209623816)

[Code 52: Enum .String() example 271](#_Toc209623817)

[Code 53: struct example 271](#_Toc209623818)

[Code 54: Map, set, slice allocation 274](#_Toc209623819)

[Code 55: Empty containers 275](#_Toc209623820)

[Code 56: nil-container example 276](#_Toc209623821)

[Code 57: Allocation domains (abstract) 278](#_Toc209623822)

[Code 58: Per-call lifetime and transfer example (happy) 281](#_Toc209623823)

[Code 59: Per-call lifetime and transfer example (sad) 281](#_Toc209623824)

Tables

[Table 1: Runtime Error Classes 58](#_Toc209623825)

[Table 2 : AMI Keywords 201](#_Toc209623826)

[Table 3: Reserved Words 202](#_Toc209623827)

[Table 4: Literal Formatting 204](#_Toc209623828)

[Table 5: Operators, Symbols, and Delimiters 205](#_Toc209623829)

[Table 6: AMI Ingress Node Required Attributes 225](#_Toc209623830)

[Table 7: Worker Function Dispositions 230](#_Toc209623831)

[Table 8: EdgeSpecifier Attributes 231](#_Toc209623832)

[Table 9: Worker Interface Edge Methods 232](#_Toc209623833)

[Table 10: AMI Collect() node attributes 242](#_Toc209623834)

[Table 11: Egress Node Methods 247](#_Toc209623835)

[Table 12: Node-State Table Methods 249](#_Toc209623836)

[Table 13: AMI toolchain 292](#_Toc209623837)

[Table 14: ami exit codes 292](#_Toc209623838)

[Table 15: ami global flags (all subcommands) 292](#_Toc209623839)

Figures

[Figure 1: A simple POP Graph 11](#_Toc209623840)

[Figure 2: Composable POP Pipeline Segments 12](#_Toc209623841)

[Figure 3: POP Data Pipeline 14](#_Toc209623842)

[Figure 4: Multiple Entrypoint POP Program 33](#_Toc209623843)

[Figure 5: Ingress Node as Entrypoint 34](#_Toc209623844)

[Figure 6: Node Identity and Metadata Lineage 44](#_Toc209623845)

[Figure 7: Event Lifecycle 51](#_Toc209623846)

[Figure 8: Pipeline Loops 86](#_Toc209623847)

[Figure 9: Concurrency Balancing Problem 113](#_Toc209623848)

[Figure 10: Worker Function Signature 125](#_Toc209623849)

[Figure 11: FanOut to Logs Example 154](#_Toc209623850)

[Figure 12: Package Import versioning example 173](#_Toc209623851)

[Figure 13: Complex Boolean Package Import Rules 173](#_Toc209623852)

[Figure 14: A Simple Pipeline Graph Visualization 223](#_Toc209623853)

[Figure 15: Node-chained pipeline example 223](#_Toc209623854)

[Figure 16: Ingress Node Topology 224](#_Toc209623855)

[Figure 17: Ingress Runtime Dispatch 229](#_Toc209623856)

[Figure 18: Edge Anatomy 231](#_Toc209623857)

[Figure 19: Transform Node Topology 234](#_Toc209623858)

[Figure 20: FanOut Node Topology 236](#_Toc209623859)

[Figure 21: Collect Node Topology 240](#_Toc209623860)

[Figure 22: Collect Data Flow 241](#_Toc209623861)

[Figure 23: edge.MultiPath(...) topology 243](#_Toc209623862)

[Figure 24: Egress Node Topology 245](#_Toc209623863)

[Figure 25: Node-State Record Structure 250](#_Toc209623864)

[Figure 26: Node State Record Types 251](#_Toc209623865)

[Figure 27: Dynamic memory container topology 274](#_Toc209623866)

[Figure 28: Empty Container 276](#_Toc209623867)

[Figure 29: Memory Allocation Domains 278](#_Toc209623868)

[Figure 30: Worker VM Lifecycle 289](#_Toc209623869)

Bibliography

Abadi, M., Cardelli, L., Pierce, B. C., & Plotkin, G. D. (1991). Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems.* *13(2)*, pp. 237–268. Association of Computing Machinery.

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and Interpretation of Computer Programs, 2nd edition.* MIT Press.

Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press.

Aho, A. V., Lam, M. S., Sethy, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques and Tools.* Addison Wesley.

Aiken, A., Fähndrich, M., & Levien, R. (1995). *Better static memory management: Improving region-based analysis of higher-order languages (Tech. Rep. UCB/CSD-95-866).* Retrieved from EECS Department, UC Berkeley.: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1995/5202.htm

*Apache Avro 1.7.7 Specification.* (2014). Retrieved from Apache Avro : https://avro.apache.org/docs/1.7.7/spec.html

Apache Software Foundation. (n.d.). *APR memory allocation (apr\_pools.h).* Retrieved from Apache Portable Runtime 1.7: https://apr.apache.org/docs/apr/1.7/apr\_\_pools\_8h.html?utm\_source=chatgpt.com

Armstrong, J. (2003). *aking reliable distributed systems in the presence of software errors (Doctoral dissertation).* Royal Institute of Technology.

Backus, J. (1978). Can Programming be Liberated From the von Neumann Style. *Communications of the ACM, 21*(8), 613-641.

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems.* O’Reilly Media.

Bjork, J., Burns, B., Fong-Jones, L., & Hochstein, L. (2020). Monitoring Distributed Systems. In B. Beyer, C. Jones, J. Petoff, & N. R. Murphy (Eds.), *The Site Reliability Workbook* (pp. 231-266). O’Reilly Media.

Borkar, V., Carey, M., Grover, R., Onose, N., & Vernica, R. (2010). Declarative systems for large-scale data processing. *Proceedings of the VLDB Endowment*, *3(1-2)dd*, pp. 1647-1654.

Bourke, T., Brun, L., & Pouzet, M. (2019). Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proceedings of the ACM on Programming Languages.* *4(POPL)*, pp. 1-29. Association of Computing Machinery.

Burns, B., & Oppenheimer, D. (2019). *Site reliability engineering: How Google runs production systems.* O’Reilly Media.

C++ Core Guidelines. (2022). *E.6: Use RAII to prevent leaks.* Retrieved from Glasp: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-raii

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin.* *38(4)*, pp. 28-38. IEEE.

Cardelli, L. (1989). Type Systems. *ACM Computing Surveys*, 21(4), 379-410.

Cardelli, L., & Wegner, P. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys.* *17(4)*, pp. 471-522. Association of Computing Machinery.

Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., & Brewer, E. (2002). Pinpoint: Problem determination in large, dynamic Internet services. *Proceedings of the International Conference on Dependable Systems and Networks*, (pp. 595-604).

Chukhman, I., Jiao, Y., Salem, H. B., & Bhattacharyya, S. S. (2016). Instrumentation-driven validation of dataflow applications. *Journal of Signal Processing Systems*, *84(3)*, pp. 383-397.

Cooper, K. D., & Torczon, L. (2011). *Engineering a compiler (2nd ed.).* Morgan Kaufmann.

Crosby, S., & Wallach, D. S. (2003). Denial of service via algorithmic complexity attacks. *Proceedings of the 12th USENIX Security Symposium* (pp. 29-44). USENIX.

Dean, J., & Barroso, L. A. (2013). The tail at scale. *Communications of the ACM*, 56(2), 74–80.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM.* *51(1)*, pp. 107-113. Association of Computing Machinery.

Decan, A., Mens, T., & Grosjean, P. (2017, 10 13). *An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems*. Retrieved from arxiv.org: https://arxiv.org/abs/1710.04936

Dennis, J. B., & Misunas, D. P. (1975). A preliminary architecture for a basic data-flow processor. *Proceedings of the 2nd Annual Symposium on Computer Architecture* (pp. 126-132). ACM.

Digibee. (2023). *Error handling strategy in event-driven integrations.* Retrieved from Digibee Documentation: https://docs.digibee.com/documentation/resources/use-cases/error-handling-strategy-in-event-driven-integrations

Donovan, A. A., & Kernighan, B. W. (2015). *The Go Programming Language.* Addison-Wesley.

Fowler, M. (2012). *Patterns of enterprise application architecture.* Addison-Wesley.

Gandhi, A., Harchol-Balter, M., Das, R., & Lefurgy, C. (2012). Optimal power allocation in server farms. *ACM SIGMETRICS Performance Evaluation Review*, 40(1), 157–168.

Gray, J., & Reuter, A. (1993). *Transaction processing: Concepts and techniques.* Morgan Kaufmann.

Ha, S., & Lee, E. A. (1997). Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers, 56*(7), 768-778.

Hardy, N. (1988). The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, pp. 22(4) 36-38.

Harper, R. (2016). *Practical Foundations for Programming Languages (2nd Edition).* Cambridge University Press.

Hellerstein, J. M. (2010). The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Record*, pp. 39(1), 5-19.

Hennessy, J. L., & Patterson, D. A. (2019). A new golden age for computer architecture. *Communications of the ACM.* *62(2)*, pp. 48-60. Association of Computing Machinery.

Herlihy, M., & Moss, J. E. (1993). Transactional memory: Architectural support for lock-free data structures. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, (pp. 289-300).

Herlihy, M., & Shavit, N. (2012). *The art of multiprocessor programming (Revised 1st ed.).* Morgan Kaufmann.

Hohpe, G., & Woolf, B. (2004). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, pp. 21(3), 359–411.

Humble, J., & Farley, D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation.* Addison-Wesley.

International Organization for Standardization. (2014). *ISO/IEC 9834-8:2014 — Information technology — Procedures for the operation of object identifier registration authorities — Part 8: Generation of universally unique identifiers (UUIDs) and their use in object identifiers.* Retrieved from iso.org: https://www.iso.org/standard/62795.html

Kaufman, C., Perlman, R., & Speciner, M. (2002). *Network Security: Private Communication in a Public World (2nd ed.).* Prentice Hall.

Kennedy, A., & Syme, D. (2001). *Design and Implementation of Generics for the .NET Common Language Runtime*. Retrieved from Microsoft Research: . https://www.microsoft.com/en-us/research/wp-content/uploads/2001/01/designandimplementationofgenerics .pdf

Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *Computer, 36(1)*, pp. 41-50.

Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations.* IT Revolution.

Klabnik, S., & Nichols, C. (2018). *The Rust Programming Language.* No Starch Press.

Kleppmann, M. (2017). *Designing Data-Intensive Applications.* O'Reilly Media.

Krabbe, G. (2019). Practical Instrumentation for Observability. *Proceedings of SREcon19 Asia.* USENIX Association.

Kreps, J. (2014). *Kafka: a distributed messaging system for log processing.* LinkedIn Engineering.

Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka: A Distributed Messaging System for Log Processing. *Proceedings of the NetDB Workshop*, (pp. 1-7).

Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM.* *21(7)*, pp. 558-565. Association of Computing Machinery.

Lattner, C., & Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. *Proceedings of the International Symposium on Code Generation and Optimization*, 75-86.

Leach, P., Mealling, M., & Salz, R. (2005). *A universally unique identifier (UUID) URN namespace (RFC 4122).* Retrieved from RFC Editor: https://www.rfc-editor.org/rfc/rfc4122

Lee, E. A. (2006). The problem with threads. *Computer*, pp. 39(5), 33–42.

Lee, E. A., & Bier, J. C. (1990). Architectures for statically scheduled dataflow. *Journal of Parallel and Distributed Computing, 10*(4), 333-348.

Lee, E. A., & Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE* (pp. 773-801). IEEE.

Leroy, X. (1992). *Polymorphic typing of an algorithmic language.* INRIA.

Levy, H. M. (1984). *Capability-Based Computer Systems.* Digital Press.

Lin, J., Gancher, D., & Parno, B. (2023). WaveCert: Formal compiler validation for asynchronous dataflow programs. *Proceedings of the ACM on Programming Languages.* *7(OOPSLA2)*, pp. 1-27. ACM.

Liu, C. L., & Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM, 20*(1), pp. 46-61.

Mace, J., Roelke, R., & Fonseca, R. (2015). Pivot tracing: Dynamic causal monitoring for distributed systems. *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)* (pp. 378-393). Association of Computing Machinery.

Majors, C. (2018). *Observability engineering: Achieving production excellence.* O’Reilly Media.

Majors, C. (2025, 06 19). *In Praise of “Normal” Engineers* . Retrieved from Charity.wtf: https://charity.wtf/2025/06/19/in-praise-of-normal-engineers/

Meyer, B. (1997). *Object-oriented software construction (2nd ed.).* Prentice Hall.

Microsoft. (2018, 02 21). *SecureZeroMemory function.* Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/previous-versions/windows/desktop/legacy/aa366877(v=vs.85

Miller, B. P. (1986). Exception handling in programming languages. *ACM Computing Surveys* (pp. 18(1), 87-110). Association of Computing Machinery.

Muchnick, S. S. (1997). *Advanced compiler design and implementation.* Morgan Kaufmann.

Newman, S. (2015). *Building microservices: Designing fine-grained systems.* O’Reilly Media.

OpenBSD Project. (n.d.). *bzero(3): explicit\_bzero — write zeroes to a byte string (OpenBSD 7.1).* Retrieved from OpenBSD Manual Pages: https://man.openbsd.org/OpenBSD-7.1/bzero.3

Ousterhout, J. (2018). *A philosophy of software design.* Yaknyam Press.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, pp. 15(12) 1053-1058.

Parr, T. (2013). *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf.

Peyton Jones, S. (2003). *Haskell 98 language and libraries: The revised report.* Cambridge University Press.

Pierce, B. C. (2002). *Types and programming languages.* MIT Press.

Pike, R. C. (2012). *Go concurrency patterns: Pipeline or concurrency?* Retrieved from Google I/O talk.: https://go.dev/talks/2012/concurrency

Preston-Werner, T. (n.d.). *Semantic Versioning 2.0.0.* Retrieved from Semantic Versioning: https://semver.org

Qi, Q. (2024, April 30). An Empirical Study on Downstream Dependency Package Groups in Software Packaging Ecosystems. (S. Hussain, Ed.) *IET Research Journal*.

Ray, B., & Kumar, A. (2007). An overview of dataflow programming paradigms. *International Journal of Software Engineering and Knowledge Engineering, 17*(6), 775-803.

Rust Project. (n.d.). *References and borrowing. In The Rust Programming Language.* Retrieved from Rust Documentation: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Rust Project. (n.d.). *What is ownership? In The Rust Programming Language.* Retrieved from Rust Documentation: https://doc.rust-lang.org/stable/book/ch04-01-what-is-ownership.html?utm\_source=chatgpt.com

Saltzer, J. H., & Kaashoek, M. F. (2009). *Principles of Computer System Design: An Introduction.* MIT Press.

Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE.* *63(9)*, pp. 1278-1308. IEEE.

Schneider, F. B. (2000). Enforceable security policies. *ACM Transactions on Information and System Security.* *3(1)*, pp. 30-50. Association of Computing Machinery.

Scott, M. L. (2016). *In Programming Language Pragmatics.* Morgan Kaufmann.

SEI CERT. (2025, 08 31). *MEM03-C. Clear sensitive information stored in reusable resources.* Retrieved from wiki.sei.cmu.edu: https://wiki.sei.cmu.edu/confluence/display/c/MEM03-C.%2BClear%2Bsensitive%2Binformation%2Bstored%2Bin%2Breusable%2Bresources

Skeen, D. (1983). *A quorum-based commit protocol.* Berkeley: University of California.

Stroustrup, B. (2003). *The C++ Programming Language (4th ed.).* Pearson Deutschland GmbH.

Syverson, P. (1994). A taxonomy of replay attacks. *Computer Security Foundations Workshop VII.* IEEE.

Tofte, M., & Talpin, J. P. (1997). Region-based memory management. *Information and Computation*, pp. 132(2), 109–176.

Weiskamp, K., Heiny, L., & Flamig, B. (1991). *Object-Oriented Programming with Turbo C++.* Wiley.

Welsh, M., Culler, D., & Brewer, E. (2001). SEDA: An architecture for well‑conditioned, scalable Internet services. *Eighth ACM Symposium on Operating Systems Principles.* *4(7)*, pp. 44-51. University of California at Berkeley.

Yusupbekov, H., & Mamatov, A. (2016). Event‑driven programming as a way to unify control statements. *International Journal of Open Information Technologies, 4*(7), 44-51.

Zaharia, M., Mosharaf, C., Das, T., Ankur, D., Ma, J., McCauley, M., . . . Stoica, I. (2012). *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.* University of California, Berkeley.

1. This rule only pertains to the declarative pipeline definition. Imperative worker function code may span many packages. [↑](#footnote-ref-1)
2. This also forces the package (and pipeline segment) as a modular, testable unit for test-driven development of software in the POP model. [↑](#footnote-ref-2)
3. The theory thus far suggests rich metadata should be a constant flow. But practical reality may require that this rich metadata be disabled for performance. A language implementing POP should weight this consideration and possibly allow the programmer to tune or disable event meta data production, where the compiler can optimize out unnecessary overhead. [↑](#footnote-ref-3)
4. Error<E> is shorthand for Error<Event<T>>. [↑](#footnote-ref-4)
5. The security practitioner may use these complex rules, for example, to exclude unpatched and vulnerable package versions. [↑](#footnote-ref-5)
6. Author’s note: The language is more than an acronym. Its name borrows from the name of the only person who listened to my ideas in 1992, including the idea for what became POP and AMI. [↑](#footnote-ref-6)