# Pipeline Semantics

This document describes the inner workings of a Conduit pipeline, its structure, and behavior. It also describes a Conduit message and its lifecycle as it flows through the pipeline.

## Pipeline structure

A Conduit pipeline is a directed acyclic graph of nodes. Each node runs in its own goroutine and is connected to other nodes via unbuffered Go channels that can transmit messages. In theory, we could create arbitrarily complex graphs of nodes, but for the sake of a simpler API we expose the ability to create graphs with the following structure:

In the diagram above we see 7 sections:

- Source connectors
- 
    - represents the code that communicates with a 3rd party system and continuously fetches records
- and sends them to Conduit (e.g.kafka connector
- ). Every source
- connector is managed by a source node that receives records, wraps them in a message, and sends them downstream to the
- next node. A pipeline requires at least one source connector to be runnable.
- Source processors
- 
    - these processors only receive messages originating at a specific source connector. Source
- processors are created by specifying the corresponding source connector as the parent entity. Source processors are not
- required for starting a pipeline.
- Fan-in node
- 
    - this node is essentially a multiplexer that receives messages produced by all source connectors and
- sends them into one output channel. The order of messages coming from different connectors is nondeterministic. A
- fan-in node is automatically created for all pipelines.
- Pipeline processors
- 
    - these processors receive all messages that flow through the pipeline, regardless of the
- source or destination. Pipeline processors are created by specifying the pipeline as the parent entity. Pipeline processors
- are not required for starting a pipeline.
- Fan-out node
- 
    - this node is the counterpart to the fan-in node and acts as a demultiplexer that sends messages
- coming from a single input to multiple downstream nodes (one for each destination). The fan-out node does not buffer
- messages, instead, it waits for a message to be sent to all downstream nodes before processing the next message (seebackpressure
- ). A fan-out node is automatically created for all pipelines.
- Destination processors
- 
    - these processors receive only messages that are meant to be sent to a specific
- destination connector. Destination processors are created by specifying the corresponding destination connector as the
- parent entity. Destination processors are not required for starting a pipeline.
- Destination connectors
- 
    - represents the code that communicates with a 3rd party system and continuously receives
- records from Conduit and writes them to the destination (e.gkafka connector
- ). Every destination connector is managed by a
- destination node that receives records and sends them to the connector. A pipeline requires at least one destination
- connector to be runnable.

There are additional internal nodes that Conduit adds to each pipeline not shown in the diagram, as they are inconsequential for the purpose of this document (e.g. nodes for gathering metrics, nodes for managing acknowledgments, etc.).

## Message

A message is a wrapper around a record that manages the record's lifecycle as it flows through the pipeline. Messages are created in source nodes when they receive records from the source connector, and they are passed down the pipeline

between nodes until they are acked or nacked. Nodes are only allowed to hold a reference to a single message at a time, meaning that they need to pass the message to the next node before taking another message[1]. This also means there is no explicit buffer in Conduit, a pipeline can only hold only as many messages as there are nodes in the pipeline (seebackpressure for more information).

[1]This might change in the future if we decide to add support for multi-message transforms.

### Message states

A message can be in one of 3 states:

- Open
- 
    - all messages start in the open state. This means the message is currently in processing, either by a node
- or a destination connector. A pipeline won't stop until all messages transition from the open state into one of the
- other two states.
- Acked
- 
    - a message was successfully processed and acknowledged. This can be done either by a processor (e.g. it
- filtered the message out) or by a destination. If a pipeline contains multiple destinations, the message needs to be
- acknowledged by all destinations before it is marked as acked. Acks are propagated back to the source connector and
- can be used to advance the position in the source system if applicable.
- Nacked
- 
    - the processing of the message failed and resulted in an error, so the message was negatively
- acknowledged. This can be done either by a processor (e.g. a transform failed) or by a destination. If a pipeline
- contains multiple destinations, the message needs to be negatively acknowledged by at least one destination before it
- is marked as nacked. When a message is nacked, the message is passed to theDLQ
- handler, which
- essentially controls what happens after a message is nacked (stop pipeline, drop message and continue running or store
- message in DLQ and continue running).

Important : if a message gets nacked and the DLQ handler successfully processes the nack (e.g. stores the message in the dead-letter queue), the source connector will receive an ack as if the message was successfully processed, even though Conduit marks it internally as nacked. In other words, the source connector will receive an ack every time Conduit handled a message end-to-end and it can be safely discarded from the source.

Pipeline nodes will either leave the message open and send it to the next node for processing or ack/nack it and not send it further down the pipeline. If the ack/nack fails, the node will stop running and return an error that will consequently stop the whole pipeline. The returned error is stored in the pipeline for further inspection by the user.

### Message state change handlers

A pipeline node can register state change handlers on a message that will be called when the message state changes. This is used for example to register handlers that reroute nacked messages to a dead-letter queue or to update metrics when a message reaches the end of the pipeline. If a message state change handler returns an error, the node that triggered the ack/nack will stop running, essentially causing the whole pipeline to stop.

## Semantics

### Messages are delivered in order

Since messages are passed between nodes in channels and a node only processes one message at a time, it is guaranteed that messages from a single source connector will flow through the Conduit pipeline in the same order that was produced by that source.

There are two caveats:

- If a pipeline contains multiple source connectors, the order of two messages coming from different connectors is
- nondeterministic. Messages coming from the same source connector are still guaranteed to retain their order.
- If a dead-letter queue is configured, negatively acknowledged messages will be removed from the stream while the
- pipeline will keep running, thus impacting the order of messages.

The order guarantee only holds inside of Conduit. Once a message reaches a destination connector, it is allowed to buffer messages and batch write them to 3rd party systems. Normally the connector would retain the order, although we can't vouch for badly written connectors that don't follow this behavior.

### Messages are delivered at least once

Between pipeline restarts, it is guaranteed that any message that is processed successfully by all nodes and not filtered out will be delivered to a destination connector at least once. Multiple deliveries can occur in pipelines with multiple destinations that stopped because of a negatively acknowledged record, or pipelines where a destination negatively acknowledged a record and processed more messages after that. For this reason, we strongly recommend implementing the write operation of a destination connector in an idempotent way (if possible).

The delivery guarantee can be changed to "at most once" by adding a dead-letter queue handler that drops unsuccessfully processed messages.

### Acks are delivered in order

Conduit ensures that acknowledgments are sent to the source connector in the exact same order as records produced by the connector. This guarantee still holds, even if a badly implemented destination connector acknowledges records in a different order, or if a processor filters out a record (i.e. acks the message) while a message that came before it is still being processed.

### Acks are delivered at most once

Acknowledgments are sent back to the source connector at most once. This means that if a message gets negatively acknowledged and is not successfully processed by a DLQ handler, the acknowledgment for that message won't be delivered to the source connector. Acknowledgments of all messages produced after this message also won't be delivered to the source connector, otherwise the order delivery guarantee would be violated. The absence of an acknowledgment after the source connector teardown is initiated can be interpreted as a negative acknowledgment.

### Backpressure

The usage of unbuffered channels between nodes and the absence of explicit buffers results in backpressure. This means that the speed of the destination connector dictates the speed of the whole pipeline.

There is an implicit buffer that needs to be filled up before backpressure takes effect. The buffer is equal to the number of nodes in a pipeline - similar to how a longer garden hose holds more water, a longer Conduit pipeline can hold more messages. There are two exceptions to this rule:

- Conduit is using gRPC streams to communicate with standalone connectors, which internally buffers requests before
- sending them over the wire, thus creating another implicit buffer (we are aware of this
- issue:#211
- ).
- Destination connectors are allowed to collect multiple records and write them in batches to the destination, which
- creates a buffer that depends on the connector implementation.

If there are multiple destinations the fanout node won't fetch a new message from the upstream node until the current message was successfully sent to all downstream nodes (this doesn't mean the message was necessarily processed, just that all downstream nodes received the message). As a consequence, the speed of the pipeline will be throttled to accommodate the abilities of the slowest destination connector.

### Dead-letter queue

Messages that get negatively acknowledged can be rerouted to another destination called a dead-letter queue (DLQ) where they are stored and can be reprocessed at a later point in time after manual intervention. If rerouting is set up and the message successfully reaches the DLQ the message will be internally nacked, but an acknowledgment will be sent to the source connector since Conduit handled the message and it can be discarded from the source. The user has the option to configure a DLQ that simply logs a warning and drops messages to achieve "at most once" delivery guarantees.

### Pipeline stop

A pipeline can be stopped in two ways - either it's stopped gracefully or forcefully.

- A graceful stop is initiated either by Conduit shutting down or by the user requesting the pipeline to stop. Only the
- source connector nodes will receive the signal to stop running. The source nodes will stop running and close their
- outgoing channels, notifying the downstream nodes that there will be no more messages. This behavior propagates down
- the pipeline until the last node stops running. Any messages that were being processed while the pipeline received a
- stop signal will be processed normally and written to all destinations.
- A forceful stop is initiated when a node stops running because it experienced an unrecoverable error (e.g. it nacked a
- message and received an error because no DLQ is configured, or the connector plugin returned an unexpected error). In
- that case, the context that is shared by all nodes will get canceled, signaling to all nodes simultaneously that they
- should stop running as soon as possible. Messages that are in the pipeline won't be drained, instead, they are dropped

- and will be requested from the source again once the pipeline is restarted. The error returned from the first node
- that failed will be stored in the pipeline and can be retrieved through the API

Edit this page Previous OpenCDC record Next Stream Inspector